Repräsentation und Generalisierung von diskreten Ereignisabläufen in Abhängigkeit von Multisensormustern

Inauguraldissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

vorgelegt von

Yorck Oliver von Collani

an der Technischen Fakultät
der Universität Bielefeld

Gedruckt auf alterungsbeständigem Papier nach DIN-ISO 9706

Für Stephanie und Tim

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit in der Technischen Fakultät der Universität Bielefeld. Die Arbeit wurde durch die Deutsche Forschungsgemeinschaft (DFG) innerhalb des Sonderforschungsbereiches 360 *Situierte Künstliche Kommunikatoren*, im Rahmen des Teilprojektes D5 *Handlungs- und Wahrnehmungsautonomie* gefördert.

Mein erster Dank gilt Prof. Dr.-Ing. Alois Knoll für die Erstellung des Gutachtens und für die Möglichkeit, die Promotion in seiner Arbeitsgruppe durchgeführt haben zu können. Ebenso danke ich PD Dr. Jianwei Zhang für die Erstellung des Gutachtens und die fachliche wie persönliche Unterstützung und Förderung während meiner Arbeit. Weiterhin möchte ich mich bei Prof. Dr. Peter Ladkin für die Gutachtertätigkeit, die Durchsicht der Arbeit und die damit verbundenen wertvollen Anregungen bedanken.

Besonderer Dank sei an diejenigen gerichtet, die mich bei der Durchsicht der Arbeit unterstützt und mir mit konstruktiven Anregungen und kritischen Hinweisen wertvolle Hilfe geleistet haben, an Dr. Jochen Ferch und Prof. Dr. Elart von Collani und Dr.-Ing. Thorsten Graf. Weiterhin möchte ich Markus Ferch für die wertvollen, anregenden und motivierenden Diskussionen danken. Nicht zuletzt geht mein Dank an Torsten Scherer für den Erhalt und Pflege der Roboterhard- und software und an Angelika Deister und Jutta Berges für das Eindämmen von orthographischen Fehlern.

Zuletzt möchte ich mich bei meiner Familie für die große Unterstützung bedanken, insbesondere bei meiner Frau Stephanie, ohne deren Rückhalt und Verständnis die Entstehung der Arbeit nicht vorstellbar gewesen wäre.

Bielefeld, Dezember 2001 *Yorck von Collani*

Inhaltsverzeichnis

1 Einführung **9**
- 1.1 Motivation . . . 9
- 1.2 Umfang der Arbeit . . . 10
- 1.3 Aufbau und Kapitelübersicht . . . 11

2 Stand der Forschung **13**
- 2.1 Architekturen . . . 13
  - 2.1.1 Deliberative Architekturen . . . 13
  - 2.1.2 Reaktive Architekturen . . . 14
  - 2.1.3 Komplexitätsreduzierung . . . 15
- 2.2 Repräsentation von Montagewissen . . . 18
  - 2.2.1 Notation . . . 19
  - 2.2.2 Repräsentationen . . . 22
  - 2.2.3 Weiterverarbeitung auf Montagesequenzen . . . 27
- 2.3 Roboterprogrammierung über Instruktionen . . . 28

3 Integrierte Robotersteuerung **31**
- 3.1 Übersicht . . . 31
- 3.2 Definitionen . . . 32
- 3.3 Systemaufbau . . . 33
  - 3.3.1 Module / Modulemanagement . . . 34
  - 3.3.2 Sequenzen / Skriptinterpreter . . . 41
  - 3.3.3 Ausnahme- und Fehlerbehandlung . . . 45
  - 3.3.4 Sensorzugang . . . 47
- 3.4 Implementationshardware . . . 48
- 3.5 Zusammenfassung . . . 49

4 Kompakte Darstellung von Sensormustern **53**
- 4.1 B-Spline Fuzzy Regler . . . 53
  - 4.1.1 B-Spline Basisfunktionen . . . 53
  - 4.1.2 Sensordatenfusion mit einem B-Spline Fuzzy Regler . . . 59
- 4.2 Hauptkomponentenanalyse . . . 60
- 4.3 Output Relevant Feature (ORF) . . . 66

4.4 Sensordatenfusion 67
4.5 Beispiel: Erkennen der Orientierung einer Schraube 68
4.6 Zusammenfassung 74

5 Diskrete Ereignisabläufe 77
5.1 Generierung und Repräsentation 79
5.1.1 Interne Repräsentation des Roboterzustandes 80
5.1.2 Sensortrajektorie 82
5.2 Generierung bei Ausnahmen 83
5.3 Zusammenfassung 87

6 Generalisierung 89
6.1 Arten der Generalisierung 89
6.2 Zusammenfassen von Sequenzen 90
6.2.1 Vergleichsmethode 90
6.2.2 Erzeugen von Verzweigungen 92
6.2.3 Auswertung von Sensormustern 100
6.2.4 Retrieval 107
6.3 Modifikation 108
6.4 Kognitive Aspekte 110
6.5 Zusammenfassung 113

7 Gesamtbeispiel 115
7.1 Zielaggregate 115
7.2 Verwendete Module und Skripte 116
7.3 Bau des Leitwerks 119
7.4 Bau des Rumpfes 123
7.5 Zusammenfassung 127

8 Zusammenfassung und Ausblick 131
8.1 OPERA 131
8.2 Lernen und Generalisieren von Montagesequenzen 135

A Technische Aspekte von *OPERA* 139
A.1 Benutzeroberfläche von *OPERA* 139
A.2 Programmierung eines *OPERA*-Moduls 148
A.2.1 Häufig benötigte Klassen 157
A.2.2 Threads 164
A.2.3 Übersetzen 164
A.2.4 Portierung auf andere Robotersteuerungen 166

B OCCL **167**
B.1 Ziel der Library 167
B.2 Überblick über die Realisierung 168
B.3 Beispielquelltext 169
B.4 Einschränkungen 170

C Anwendung der Reduktionsverfahren **171**

D Petri-Netz **177**

E Veröffentlichungen **179**

Abbildungsverzeichnis **180**

Tabellenverzeichnis **185**

Index **187**

Literaturverzeichnis **191**

Kapitel 1

Einführung

1.1 Motivation

Montage ist ein schwieriger Prozess, aber auch eines der wichtigsten Gebiete der Robotik. Das Problem der Montage besteht im Design eines rechner-gesteuerten Robotersystems, welches in der Lage ist, ein Objekt aus seinen Einzelteilen zusammenzusetzen. Während der Realisierung eines solchen Systems müssen mehrere Ziele in Einklang gebracht werden. Typische Montagesysteme haben zur Zeit folgende Schwierigkeiten:

1. Mangel an Flexibilität. Um bei der Programmierung eines Montagesystems den Arbeitsaufwand zu reduzieren, wird oft von einer wohlbekannten Umgebung ausgegangen z.B. feste Montagepositionen. Dadurch kann das System schneller eingesetzt werden, ist aber aufgrund der fehlenden Flexibilität nicht mehr in der Lage, bei größeren Änderungen der Umgebung zu arbeiten.

2. Hoher Arbeitsaufwand. Die mangelnde Flexibilität und die festen Abhängigkeiten zwischen Methoden und Algorithmen für ein spezifisches Objekt machen eine häufige und aufwendige Reprogrammierung des Systems notwendig, falls sich die Rahmenbedingungen ändern.

Es ist daher leicht einzusehen, dass ein Hauptdesignziel eines modernen Robotersystems hinreichende Flexibilität ist, wobei verschiedene Arten von Flexibilität unterschieden werden können: Die *externe* Flexibilität ist die Fähigkeit des Systems, sich an unterschiedliche Objekte und ihrer Teilobjekte adaptieren zu können. Als *interne* Flexibilität wird die Fähigkeit des Systems bezeichnet, mit unterschiedlichen Randbedingungen und Toleranzen der Bauteile umgehen zu können. Die Anforderungen an ein flexibles Robotermontagesystem lassen sich daher wie folgt zusammenfassen:

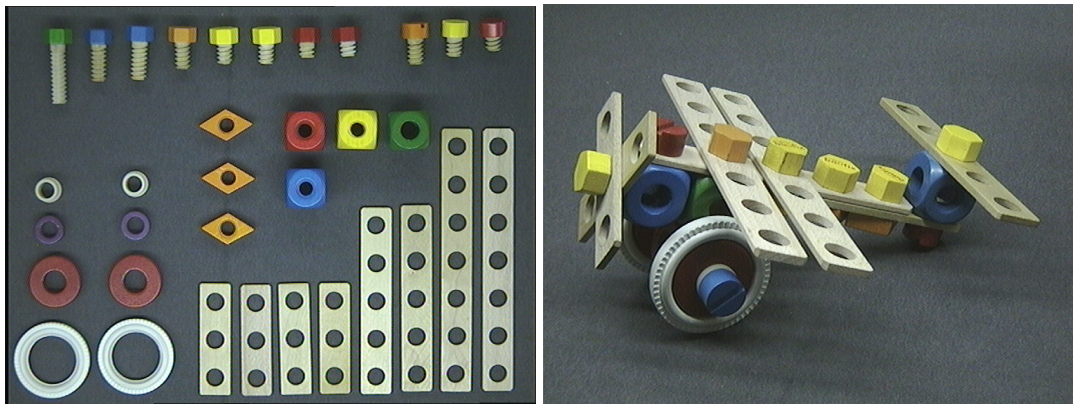
- Das System muss einfach zu reprogrammieren sein.

- Das System muss von verschiedenen Gruppen von Endbenutzern zu gebrauchen sein.

- Das System muss mit verschiedenen Hard- und Softwarekomponenten umgehen können.
- Das System sollte verschiedene Arten von Sensoren integrieren und verwenden.
- Es gibt keine Programmiersprache mit deren Hilfe alle Teile eines Systems gleich gut zu implementieren sind. Daher wäre es wünschenswert, wenn sich in das Montagesystem Module mit unterschiedlichen Sprachen integrieren ließen.

1.2 Umfang der Arbeit

Die vorliegende Arbeit befasst sich hauptsächlich mit dem ersten und dem vorletzten der genannten Punkte, wobei die anderen Punkte zwar mit berücksichtigt werden, aber etwas im Hintergrund stehen. Es wird ein Montagesystem und ein auf diesem System aufbauendes Lernverfahren vorgestellt, welches viele der oben beschriebenen Gesichtspunkte realisiert. Das Lernverfahren dient zum interaktiven Lernen von Montagesequenzen.



(a) Bauteile des Spielzeugsystems *Baufix*. (b) Baufix-Flugzeug.

Abbildung 1.1: *Verwendete Bauteile und angestrebtes Zielaggregat des SFB's 360.*

Der Hintergrund für die folgende Arbeit stellt der Sonderforschungsbereich 360 dar, dessen Ziel die Entwicklung eines künstlichen situierten Agenten zur Erforschung von linguistischen und kognitiven Eigenschaften menschlicher Intelligenz ist. Als Bespielszenario wurde die Montage eines Flugzeuges (Abb. 1.1 (b)) aus Bauteilen des Spielzeugsystems *Baufix* (Abb. 1.1 (a)) genommen, in dem ein *Instrukteur* interaktiv mit dem Agenten die Montage durchführt. Der Instrukteur beschreibt mit Hilfe sprachlicher Äußerungen und Gesten den Montagevorgang, der vom Montagesystem *online* durchgeführt

wird. Dieses System soll nach einer abgeschlossenen Montage nicht nur in der Lage sein diese exakt, sondern sie auch in leicht abgewandelter Form zu wiederholen.

Es wird eine Methode zur Repräsentation und Wiederholung von Montagesequenzen vorgestellt, und darüber hinaus der Aspekt der Generalisierung von Montagevorgängen berücksichtigt. Insbesondere wird Wert auf die Einbeziehung interaktiver Aspekte (Mensch-Maschine-Kommunikation) und **realer** Montagevorgänge gelegt, welches die Verarbeitung von realen Sensordaten unterschiedlicher Art zur Folge hat. Ein weiterer zentraler Punkt ist die Parametrisierung gelernter Handlungen, um die Durchführung von *modifizierten Anweisungen* zu realisieren. Es sind folgende Probleme gelöst worden:

- Implementation einer Kontrollarchitektur für Montagevorgänge, ohne sich generell auf das Lernverfahren als einzigen Anwendungsfall festzulegen.
- Einbeziehung eines realen Robotersystems mit seinen vielfältigen Unsicherheiten.
- Berücksichtigung realer hochdimensionaler und verrauschter Sensordaten.
- Verarbeitung von Sensormustern unbekannter Struktur.
- Realisierung eines Verfahrens zur Akquirierung und Generalisierung von Montagewissen mittels Lernen.

1.3 Aufbau und Kapitelübersicht

Die weiteren Kapitel haben folgenden Inhalt:

In Kapitel 2 wird auf die schon vorhandenen Lösungen eingegangen und der Stand der Forschung erläutert. Dabei wird aufgezeigt, welche Systeme zum Aufbau einer Robotersteuerung existieren. Diese reichen von kompletten Entwicklungsumgebungen bis hin zu Funktionsbibliotheken bestimmter Programmiersprachen und Betriebssystemen. Anschließend werden mögliche Arten der Repräsentation von Montagesequenzen für ein Robotersystem erörtert und andere Ansätze vorgestellt, Manipulatoren bestimmte Fertigkeiten ohne direkte Programmierung beizubringen.

Kapitel 3 beschreibt das Robotermontagesystem *OPERA* (**O**pen **P**rogramming **E**nvironment for **R**obot **A**pplications). Nach einer Übersicht der zentralen Eigenschaften werden Definitionen von Mengen und Funktionen gemacht, mit deren Hilfe die einzelnen Komponenten, ihre Funktionalität und Semantik beschrieben werden. Anschließend wird auf den Systemaufbau von *OPERA* eingegangen und die zentralen Komponenten werden erörtert. Es folgt die Beschreibung des Begriffes *Modul*. Ein Modul nimmt in der Konzeption von *OPERA* eine zentrale Stellung ein, da alle Funktionalitäten um die *OPERA* erweitert werden kann, in Form eines Moduls implementiert werden. Es wird die allgemeine Funktionalität beschrieben und die unterschiedlichen Modularten werden erklärt. Nach einer Beschreibung der Kommunikations- und Interaktionsmöglichkeiten eines Moduls im System wird die Anbindung eines Roboters an *OPERA* beschrieben. Es

folgt die Aufbaubeschreibung des Interpreters und der Montageskripte mit einer Definition der Instruktionssemantik. Es wird auf die Mechanismen zur Fehler- und Ausnahmebehandlung, der Sensorintegration, der verwendeten Hardware eingegangen und mit einer Zusammenfassung das Kapitel abgeschlossen.

Im Kapitel 4 wird eine Methode für den Umgang mit Sensormustern unbekannter Struktur vorgestellt. Als erstes werden die Grundlagen der verwendeten Verfahren und der B-Spline Fuzzy Regler als allgemeiner Funktionsapproximator erläutert und dessen Anwendung in Verbindung mit der *PCA* (principal component analyse) und der *ORF* (output relevant feature) erklärt. Desweiteren wird ein Verfahren zur Sensordatenfusion vorgestellt, welches auf den vorher vorgestellten Verfahren beruht.

Kapitel 5 beschäftigt sich mit der Repräsentation von Ereignisabläufen und ihrer Generierung in Normal- und Ausnahmefällen.

In Kapitel 6 wird die Generalisierung der gespeicherten Ereignisabläufe behandelt. Es werden zwei Arten von Generalisierung eingeführt; die Abwandlung von Montagesequenzen durch die Verwendung anderer Montageteile und die selbständige Auswahl von Sequenzsträngen während einer Montage. Es wird ein Verfahren zur Fusion von Montagesequenzen und zur Generierung von Verzweigungen des Montageprozesses erläutert. Anschließend wird das Lernverfahren aus kognitiver Sicht betrachtet und Bezüge zu unterschiedlichen Gedächtnisformen hergestellt.

In Kapitel 7 wird anhand der Montage eines Leitwerks und eines Flugzeugrumpfes aus Bauteilen des Spielzeugs *Baufix* die Funktionalität des Gesamtsystems demonstriert.

In Kapitel 8 folgt eine Zusammenfassung der vorgestellten Kontrollarchitektur und des Lernverfahrens. Es werden die Unterschiede zu anderen Ansätzen erläutert und mögliche Erweiterungen des Verfahrens aufgezeigt.

Im Anhang wird auf die technischen Aspekte von *OPERA* eingegangen. Es werden die Benutzerschnittstelle und die Programmierung eines Moduls in der Programmiersprache C++ erklärt und wichtige Klassen beschrieben.

Auf der beiliegenden CD-ROM enthält die Quelltexte von *OPERA* und den Reduktionsalgorithmen sowie die Quelltextdokumentation.

Kapitel 2

Stand der Forschung

Die vorliegende Arbeit berührt im Wesentlichen drei Themengebiete: Architektur eines (komplexen) Montagesystems, Repräsentation und Lernen von Montagesequenzen. Es wird daher im Folgenden auf den aktuellen Stand dieser Forschungsbereiche eingegangen.

2.1 Architekturen

Zum Aufbau eines lernenden Montagesystems bedarf es einer Architektur, die die einzelnen benötigten Komponenten (Roboteransteuerung, Wissensbasen, Elementaroperationen, u.s.w.) koordiniert. Nach [CMS00] umfassen die Anforderungen an eine Software-Architektur die Ausführung von abstrakten und komplexen Zielen, die Interaktion mit komplexen und oft dynamischen Umgebungen unter Berücksichtigung von Rauschen, Ausnahmen, unvorhersehbaren Zustandswechseln und der eigenen Systemdynamik. Außerdem werden viele Roboter in kritischen Bereichen eingesetzt, so dass die Systemsicherheit eine große Rolle spielt, da Fehler in der Ausführung schwerwiegende Auswirkungen haben können. Abb. 2.1 zeigt das Spektrum [Ark98] der aktuellen Kontrollarchitekturen. Die linke Seite repräsentiert die Architekturen, die einen deliberativen Ansatz verfolgen, und die rechte Seite diejenigen, die über Verhalten gesteuert werden. Die deliberativen Ansätze basieren auf einer Modellierung der Umwelt, so dass ihre Leistungsfähigkeit davon abhängt, wie genau die Welt modelliert ist und wie konsistent die Modellierung mit der Wirklichkeit ist.

2.1.1 Deliberative Architekturen

Diese deliberativen Systeme haben viele Gemeinsamkeiten:

- Sie sind hierarchisch strukturiert, mit einer klaren Unterteilung der Funktionalitäten, ähnlich der Struktur eines Unternehmens oder einer militärischen Einheit.

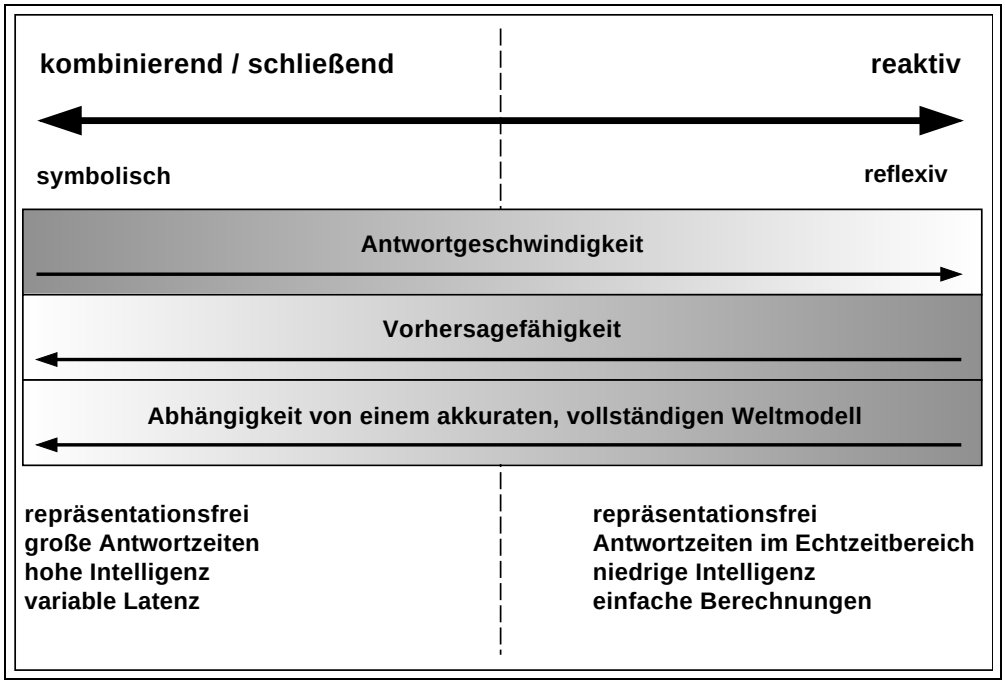


Abbildung 2.1: *Hierarchiespektrum.*

- Kommunikation erfolgt in einer vorhersehbaren und vorbestimmten Weise, vertikal zwischen den einzelnen Abstraktionsschichten. Ein horizontaler Informationsfluss zwischen Komponenten in einer Hierarchieebene findet kaum statt.
- Höhere Schichten zerlegen die Aufgabe in Teilaufgaben und geben sie an die darunterliegende Schicht zur Ausführung weiter.
- Der Planungsbereich wechselt beim Abstieg in der Hierarchie. Je niedriger die Ebene um so konkreter und zeitlich eingeschränkter ist die Planung.
- Große Abhängigkeit von der symbolischen Repräsentation des Weltmodells.

2.1.2 Reaktive Architekturen

Im Gegensatz zu diesem *Top-Down* Ansatz verfolgt der reaktive Ansatz eine *Bottom-Up* Strategie [Bro86]. Dieser Ansatz ist auf der rechten Seite des Spektrums in Abb. 2.1 repräsentiert. Die Software besteht aus einer Menge von Modulen *(behaviors)*, die gleichzeitig und fortwährend ausgeführt werden und über Kommunikationsmechanismen oder über die Umgebung miteinander interagieren. Diese Systeme sind daher einfach aufgebaut, besitzen aber eine enge Koppelung zwischen Wahrnehmung und Steuerung. Typisch sind motorische Verhalten, um schnell auf eine dynamische und wechselnde Umwelt reagieren zu können. Mit dieser Architektur ist es nicht einfach, abstrakte Zielsetzungen zu verfolgen und die Regeln, nach denen die einzelnen Module miteinander

interagieren, erschweren die Behandlung von Sicherheitsaspekten. Auch verhaltensbasierte Architekturen haben einige Gemeinsamkeiten:

- Fokussierung auf eine enge Verbindung von Wahrnehmung und Reaktion.
- Abwesenheit von symbolischer Repräsentation.
- Unterteilung in kontextabhängige Einheiten.

Neben den zuvor genannten Gemeinsamkeiten wird bei den reaktiven Architekturen auch noch in der Granularität der Verhaltensunterteilung und in der Koordination einzelner Methoden (konkurrierend oder kooperierend) unterschieden. Beispiele für solche verhaltensbasierte Architekturen sind in [Bro86, Ark87, Kae86, Mae90, Con89, Fir89] zu finden.

Eine hybride Architektur kombiniert beide vorhergehenden Ansätze und verbindet somit das Design von effizienter Steuerung auf niedrigen Komplexitätsebenen mit der Verfolgung von abstrakten Zielstellungen. Die Schnittstelle zwischen diesen beiden Ansätzen ist unter Umständen trickreich und muss genau geplant werden.

Die meisten Architekturen lassen sich somit grob in drei Kategorien einteilen: hierarchisch, verhaltensgesteuert und hybrid.

2.1.3 Komplexitätsreduzierung

Um das Problem der Komplexität größerer hierarchischer Systeme zu lösen, wird oft die bestehende Struktur modularisiert. Die Komplexität wird durch die Aufteilung in kleinere überschaubare Komponenten vermindert. Dies geschieht z.B. durch die Trennung der Datenströme bei einer Datenflussarchitektur oder bei einer funktionalen Struktur durch die Bildung von Unterfunktionen, die ihre Ergebnisse an den Funktionsaufrufer zurückgeben. Diese Unterteilung des Systems ist oft auch hierarchisch. Module einer bestimmten Ebene in der Hierarchie basieren auf anderen Modulen einer darunter liegenden Ebene.

Eine solche Architektur für Echtzeitsteuerungssysteme wird in [Alb97b] vorgestellt. Dieses Referenzarchitekturmodell besteht aus einer hierarchisierten und geschichteten Menge von Knoten, die durch ein Kommunikationsnetzwerk verbunden sind. Jeder dieser Knoten (Abb. 2.2) enthält folgende Bestandteile:

1. *Behaviour Generation* (BG)
2. *World Modelling* (WM)
3. *Sensory Processing* (SP)
4. *Value Judgement* (VJ)
5. *Knowledge Database* (KD)

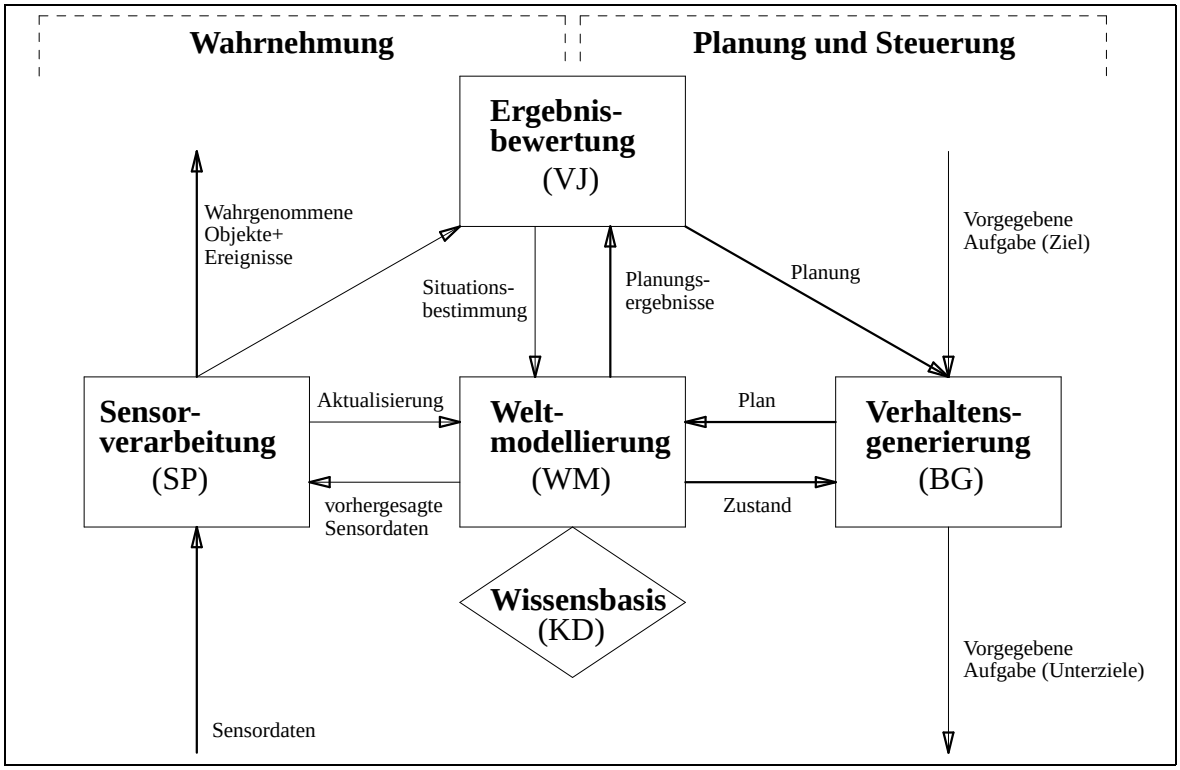


Abbildung 2.2: *Ein typischer Knoten in der* Real-time Control System *(RCS) Architektur (aus [Alb97a]).*

Die Aufgabe bzw. Anweisung für einen Knoten kommt von einem BG Prozess einer höheren Hierarchieebene. Jede Anweisung wird in Teilanweisungen zerlegt und an untergeordnete BG Prozesse weitergereicht. Ein Prozess für die Weltmodellierung WM beinhaltet eine Wissensbasis KD, die dem BG Prozess zur Abschätzung der externen Welt dient. Der Prozess zur Verarbeitung von Sensordaten SP nimmt eingehende Sensorsignale entgegen und führt Tätigkeiten wie Aufmerksamkeitssteuerung, Attributberechnung, Filterung u.s.w. durch. Ein Prozess VJ bewertet die Erwartungswerte eines vorläufigen Plans und Daten, die in die Wissenbasis aufgenommen werden sollen.

Jeder BG Prozess enthält eine Planungseinheit, die die Anweisungen entgegennimmt und daraus Teilpläne für untergeordnete BG Prozesse generiert. Die Planungseinheit erstellt dabei eine Reihe von vorläufigen Plänen, deren Resultat vom WM Prozess vorausgesagt und vom VJ bewertet wird. Der Plan mit dem besten Resultat wird in die Ausführungseinheit *(Executor)* übergeben. Für jeden untergeordneten BG Prozess existiert eine Ausführungseinheit, die die Teilanweisungen weitergibt, die Ausführung beobachtet, Fehler und Differenzen zwischen geplanten und beobachteten Situationen kompensiert und schnell auf Ausnahmesituationen reagiert. Daten der Wissensbasis machen es den Ausführungseinheiten möglich, gegenwirkende Maßnahmen zu ergreifen. SP und WM Prozesse versorgen die Wissensbasis KD mit aktuellen Daten (z.B.

Bildern, Karten, Zuständen,...), die sowohl für überlegte wie auch reflexartige Verhaltensweisen notwendig sind. Abb. 2.3 zeigt exemplarisch ein System, das nach diesem

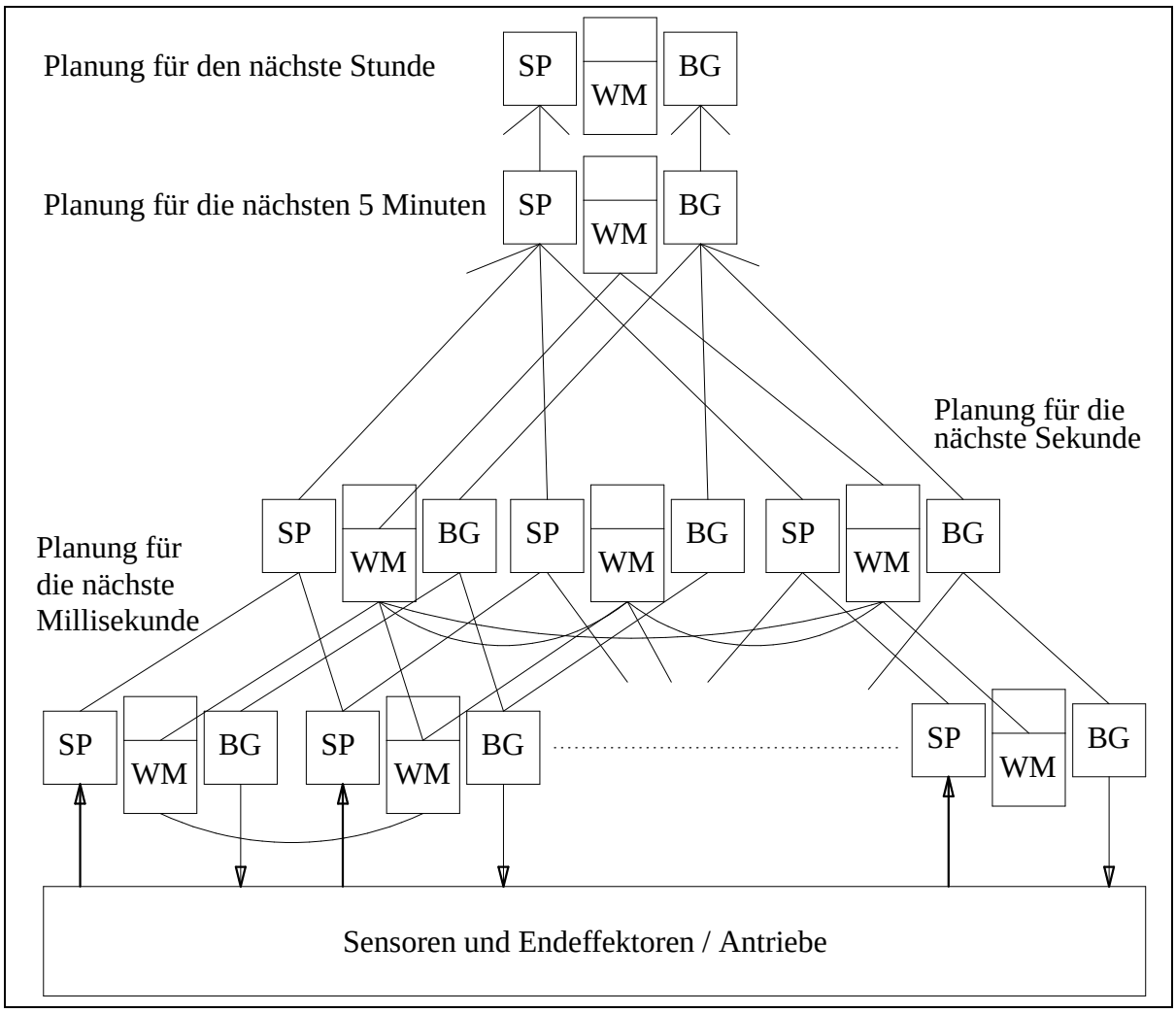


Abbildung 2.3: *Hierarchische Architektur (aus [Alb97a]).*

Modell aufgebaut ist. Die Anwendung dieser Architektur in einem unbemannten Bodenfahrzeug ist in [Alb00] beschrieben. Eine Übersicht von Projekten, die diese Architektur verwenden, ist in [Alb95] enthalten und eine Erweiterung dieser Architektur zur Sensordatenfusion ist in [HM99] vorgestellt.

Viele andere Architekturen [BKW97, Wal97, FG98] basieren auf dieser Strategie der Komplexitätsreduzierung, wobei die Vorgabe für eine Hierarchisierung unterschiedlich stark ausgeprägt und problemspezifisch ist. So wird in [BKW97] ein Hierarchie vorgeben. Das vorgestellte System besteht aus drei Schichten. Die erste Schicht (Skill Manager) besteht aus einer Menge von roboterspezifischen Basisoperationen wie Greifen, Objektverfolgung und lokale Navigation. Die darauf aufbauende Schicht (Sequenzer)

beinhaltet einen Interpreter über den die Basisoperationen reaktiv aktiviert werden. Auf der obersten Ebene befindet sich die Planungseinheit, die Ziele, Ressourcen und Zeitvorgaben koordiniert. Im Gegensatz dazu wird in [Wal97, FG98] das System über verschiedene Komponenten modularisiert. Über ein Client-Server Ansatz kommunizieren die einzelnen Komponenten unter einander, so dass erst die konkrete Applikation eine Hierarchie verwirklicht.

Ein andere Möglichkeit zur Komplexitätsreduktion eines System ist die Verwendung einer problemorientierten Programmiersprache. Eine solche Sprache verdeckt die Komplexität des verwendeten Systems und erlaubt eine aufgabenorientierte Programmierung. Sprachen, die Echtzeitaspekte berücksichtigen, sind z.B. *Subsumption language* [Bro86], *ALPHA* [Gat92] und *Skills* [RFG+97]. Ein anderer Ansatz ist die Angabe der Steuerungsstrategie über eine graphische Umgebung. Die Steuerungsprogramme werden dann automatisch aus der graphischen Beschreibung generiert [SCPCW98, ZMS97]. Außerdem existieren noch Zusätze zu vorhandenen allgemeinen Programmiersprachen (C++, Lisp), die diese Sprachen um Kontrollstrukturen auf Aufgabenebene erweitern [Gat96, SA98]. In [BKW97] basiert die Sprache des Interpreters auf LISP.

Ein weiterer wichtiger Aspekt von Robotiksystemen ist die Detektion und Behandlung von Fehlerfällen. In einigen Architekturen sind Ausnahmen nicht bekannt und jeder Systemzustand wird in gleicher Weise behandelt [Bro86]. In anderen Architekturen wird eine Ausnahme in anderer Weise behandelt. Zumeist existiert eine spezielle Instanz *(Excepetion Handler)*, die die Ausnahme behandelt, bevor mit der normalen Verarbeitung fortgefahren wird [PHH99].

2.2 Repräsentation von Montagewissen

Eine Repräsentation von Montagewissen sollte folgende Informationen beinhalten:

- Welches Objekt soll zusammengebaut werden?
- Welche Operationen sind verboten, um nicht während der Montage in einen Zustand zu gelangen, von dem aus die Montage nicht fortzuführen ist?
- Welche Operationen sind anderen vorzuziehen?

Für die Repräsentation von Montagewissen werden vorwiegend Graphstrukturen verwendet, wobei sich zwei Repräsentationstypen unterscheiden lassen: In der *impliziten* Repräsentation existiert keine direkte Abbildung von der eigentlichen Montageoperation in die symbolischen Repräsentation des zu bauenden Aggregates. Stattdessen bestehen diese Repräsentationen aus Bedingungen, die durch die Montage erfüllt werden müssen. Devanathan et al. [SD94] stellen eine temporäre Logik als ein *implizites* Repräsentationswerkzeug und eine analytische Sprache für Montagesequenzplanung vor. Dort sind die Knoten des Montagegraphs mit dem Zustand der Montage

gleichzusetzen, wobei unter dem Zustand die Aussage verstanden wird, welche der zur Verfügung stehenden Montageoperationen ausgeführt worden sind und welche nicht. In einer *expliziten* Repräsentation existiert eine direkte Abbildung von der Montageoperation in die Elemente der Repräsentation. Diese Graphen enthalten alle geometrischen Sequenzen, angefangen von einem Bauteil über Teilobjekte bis hin zum fertigen Objekt. Die Konstruktion solcher Graphen wird manuell oder automatisch durch die Demontage eines fertigen Aggregates durchgeführt. In beiden Repräsentationen bezeichnen die Knoten einen statischen Zustand des Systems und die Kanten repräsentieren einen Montageschritt.

2.2.1 Notation

Eine mechanische Montage ist das Verbinden von sich berührenden Bauteilen zu einer stabilen Einheit (siehe auch [RW91]). Dabei wird davon ausgegangen, dass

- die Bauteile starr sind und ihre Form nicht verändern.
- Bauteile verbunden sind, wenn sie sich mit mindestens einer kompatiblen Oberfläche berühren.
- die Berührung zweier kompatibler Oberflächen den Grad der Bewegungsfreiheiten beider Bauteile reduziert.

Ein Teilaggregat (auch Baugruppe) kann mit mehreren unterschiedlichen Teilaggregaten verbunden sein und kann daher auch mehrere Verbindungen besitzen.

Ein Teilaggregat ist eine nicht leere Teilmenge von Bauteilen und enthält ein Bauteil oder mehrere verbundene Bauteile. Die Geometrie eines Teilaggregates wird durch die relative Lage der Bauteile zueinander bestimmt und das Teilaggregat ist stabil, wenn die Bauteile ihre relative Position zueinander nicht spontan verändern. Eine spontane Lageänderung kann z.B. dann auftreten, wenn die Lage zweier Teilaggregate zueinander von der Orientierung des Aggregates, in dem sie montiert sind, abhängt. Ein Teilaggregat aus einem einzelnen Bauteil ist stabil. Ein Montageprozess besteht aus einer Folge von Operationen, die Teilaggregate zusammenfügen. Es wird ebenso angenommen, dass nach einer Montageoperation alle Verbindungen zwischen den montierten Bauteilen bestehen. Durch diese Voraussetzungen lässt sich ein Aggregat durch einen ungerichteten Graphen $< P, C >$ mit $P = \{p_1, \ldots, p_N\}$ als Knoten und $C = \{c_1, \ldots, c_L\}$ als Kanten repräsentieren. Jeder Knoten korrespondiert mit einem Bauteil und jede Kante mit einer Verbindung von zwei Bauteilen. Dieses relationale Baugruppenmodell von Homem De Mello [dM89] berücksichtigt keine geometrischen und physikalischen Bedingungen, um welche das Modell z.B. in [Mos00] erweitert wird. Dort wird das Modell um geometrische, physikalische und Lageinformationen erweitert, die es später erlauben durch Dekomposition der Baugruppe einen Montageplan zu erstellen.

Montagezustände

Der Zustand eines Montageprozesses ist die Konfiguration aller Bauteile zu Beginn oder zum Ende einer Montageoperation. Da nach einer Operation alle Verbindungen zu Stande gekommen sind, ist ein Montagezustand durch einen L-dimensionalen binären Vektor $\vec{x} = (x_1, \ldots, x_L)$ repräsentiert. Jede Komponente eines solchen Vektors ist entweder wahr oder falsch, je nachdem, ob die i-te Verbindung besteht oder nicht. Aus demselben Grund kann ein Montagezustand auch durch die Menge seiner Bauteile charakterisiert werden. Eine Menge von Bauteilen wird als Teilaggregat, bestehend aus diesen Bauteilen, verstanden.

Beispiel: Besteht das zu bauende Aggregat *Schiff* aus den Bauteilen `Rumpf`, `Mast` und `Segel`, so lässt sich der Anfangszustand durch

$$\{\{\texttt{Rumpf}\}, \{\texttt{Mast}\}, \{\texttt{Segel}\}\}$$

ein Zwischenzustand durch

$$\{\{\texttt{Rumpf,Mast}\}, \{\texttt{Segel}\}\}$$

und der Endzustand durch

$$\{\{\texttt{Rumpf,Mast,Segel}\}\}$$

bezeichnen.

Jeder Zustand lässt sich mit einem einfachen ungerichteten Graph $< P, C_k >$ assoziieren mit P als Menge der Bauteile und C_k ($C_k \subseteq C$) als Menge der zu Stande gekommenen Verbindungen.

Die Funktion *sa* gibt an, ob zwei Teilaggregate zusammen ein gültiges Teilaggregat ergeben. So ergibt *sa*$(\{\text{Rumpf}\}, \{\text{Mast}\}) =$ wahr und *sa*$(\{\text{Rumpf}\}, \{\text{Segel}\}) =$ falsch. Des Weiteren gibt die Funktion *st* an, ob ein Teilaggregat stabil ist oder nicht.

Montageoperationen

Existieren zwei Teilaggregate θ_i und θ_j dann ist die Vereinigung von θ_i und θ_j eine Montageoperation, wenn die Menge

$$\theta_k = \theta_i \cup \theta_j$$

ein Teilaggregat bildet. Alternativ kann eine Operation als Demontage angesehen werden, wenn das Endresultat aus zwei Teilaggregaten besteht. Daher kann eine Montageoperation als eine Menge bestehend aus zwei Untermengen definiert werden. Die eine Untermenge beinhaltet die zu verbindenden Teilaggregate und die andere die herzustellenden Verbindungen $c_i \in C$.

Eine Montageoperation ist *geometrisch* ausführbar, wenn die beiden zu montierenden Teilaggregate auf einem kollisionsfreien Pfad in Kontakt gebracht werden können.

Kommt dabei eine Verbindung zu Stande, ist die Operation *mechanisch* ausführbar. Die Funktionen *gf* und *mf* geben an, ob eine Montageoperation geometrisch bzw. mechanisch durchführbar ist.

Montagesequenzen

Besteht ein Aggregat aus N Bauteilen, dann ist die geordnete Menge $\tau_1, \ldots, \tau_{N-1}$ von Montageoperationen eine Montagesequenz, wenn:

- keine zwei Operationen existieren, die als Eingabe dieselben Teilaggregate besitzen.
- das Ergebnis der letzten Operation das vollständige Aggregat ist.
- die Eingabe einer Operation entweder ein einzelnes Bauteil oder das Resultat einer vorherigen Montageoperation ist.

Zu einer Montagesequenz $\tau_1, \ldots \tau_{N-1}$ existiert eine korrespondierende Sequenz von N Montagezuständen $s_1, \ldots, s_N$ des Montageprozesses. Der Zustand s_1 ist der Zustand in dem alle Bauteile separat und s_N in dem das fertige Aggregat vorliegt. Zwei aufeinander folgende Zustände s_i und s_{i+1} sind so beschaffen, dass die beiden Teilaggregate, die durch die Operation τ_i verbunden werden, im Zustand s_i und nicht in s_{i+1} existieren und das Montageresultat in s_{i+1} nicht in s_i. Daher kann eine Montagesequenz auch als eine geordnete Menge von Montagezuständen charakterisiert werden.

Eine Sequenz gilt als durchführbar, wenn alle Operationen $\tau_1, \ldots, \tau_{N-1}$ geometrisch und mechanisch durchführbar sind. Eine (nicht notwendigerweise durchführbare) Montagesequenz kann auf verschiedene Weise repräsentiert werden:

- Als eine geordnete Liste von Operationen. Die Länge der Liste ist $N - 1$.
- Als eine geordnete Liste von binären Vektoren. Jeder Vektor korrespondiert mit einem (nicht notwendigerweise stabilen) Zustand. Die Länge der Liste ist N.
- Als eine geordnete Liste von Mengen von Bauteilmengen. Jede Menge korrespondiert mit einem (nicht notwendigerweise stabilen) Zustand. Die Länge der Liste ist N.
- Als eine geordnete Liste von Verbindungsteilmengen. Die Länge der Liste ist $N-1$.

Diese Arten der Repräsentation haben folgende Eigenschaften:

- Für jede geordnete Liste von binären Vektoren $(\vec{x}_1, \ldots, \vec{x}_N)$ mit $\vec{x}_i = (x_{i1}, \ldots, x_{iL})$, die eine Montagesequenz repräsentiert, gilt

  $$[(j > i) \wedge (x_{ik} = \text{wahr})] \Rightarrow (x_{jk} = \text{wahr}) \ .$$

  Dies korrespondiert mit der Aussage, dass eine einmal bestehende Verbindung nicht mehr gelöst wird.

- Für eine geordnete Liste von Mengen von Bauteilmengen $(\Theta_1, \ldots, \Theta_N)$, welche eine Montagesequenz repräsentiert, gilt:

  $$[(j > i) \wedge (\theta_a \in \Theta_i)] \Rightarrow \exists\theta_b[(\theta_b \in \Theta_j) \wedge (\theta_a \subseteq \Theta_b)]$$

  Dies korrespondiert mit der Aussage, dass zwei verbundene Teilaggregate bis zum Ende der Montage verbunden bleiben.

2.2.2 Repräsentationen

Direkte Graphrepräsentation einer Montagesequenz

Ist das zu montierende Aggregat durch den Graphen $< P, C >$ definiert, kann eine direkte Graphrepräsentation verwendet werden [RW91]. Die Knoten in diesem Graphen korrespondieren mit einer stabilen Zustandsmenge aus der Montage P. Dies sind Mengen Θ aus P, so dass $\theta \in \Theta$ und θ ein stabiles Teilaggregat aus P ist.

Die Kanten dieses direkten Graphen sind geordnete Paare von Knoten. Für jede Kante existieren nur zwei Teilmengen θ_i und θ_j in der Menge des ersten Knoten, welche sich nicht in der Menge des zweiten Knoten befinden. Es existiert ebenso nur eine Teilmenge θ_k in der Menge des zweiten Knoten, welche nicht in der Menge des ersten Knoten ist, mit

$$\theta_k = \theta_i \cup \theta_j$$

Des Weiteren ist die Operation, die θ_i und θ_j verbindet, durchführbar.

Definition 2.2.1 *Der* direkte Graph einer durchführbaren Montagesequenz*, eines Aggregates, dessen Bauteilmenge P ist, ist ein gerichteter Graph $< X_P, T_P >$ mit*

$$X_P = \{\Theta | [\Theta \in \Delta(P)] \wedge [\forall\theta(\theta \in \Theta) \Rightarrow (\mathsf{sa}(\theta) \wedge \mathsf{st}(\theta))]\}$$

als Menge aller stabilen Montagezustände und

$$\begin{aligned} T_P = \{(\Theta_i, \Theta_j) \quad | \quad & [(\Theta_i, \Theta_j) \in X_P \times X_P] \quad \wedge \\ & [\mathcal{U}(\Theta_i - (\Theta_i \cap \Theta_j)) \in \Theta_j - (\Theta_i \cap \Theta_j)] \wedge \\ & [|\Theta_j - (\Theta_i \cap \Theta_j)| = 1] \wedge [|\Theta_i - (\Theta_i \cap \Theta_j)| = 2] \wedge \\ & [\mathsf{mf}(\Theta_i - (\Theta_i \cap \Theta_j))] \wedge [\mathsf{gf}(\Theta_i - (\Theta_i \cap \Theta_j))]\} \end{aligned}$$

$\Delta(P)$ bezeichnet alle Mengen von Mengen von P und $\mathcal{U}(\{A, B, \ldots, Z\}) = A \cup B \cup \cdots \cup Z$. Abb. 2.4 zeigt ein Beispiel eines solchen Graphen.

AND/OR Graph

Ein AND/OR Graph kann ebenfalls zur expliziten Repräsentation von Montagesequenzen verwendet werden [Nil80, RW91]. Die Knoten in einem AND/OR Graph sind Teilmengen von P und charakterisieren ein stabiles Teilaggregat. Der Wurzelknoten repräsentiert die fertig montierte Baugruppe. Die Hyperkanten korrespondieren mit einer

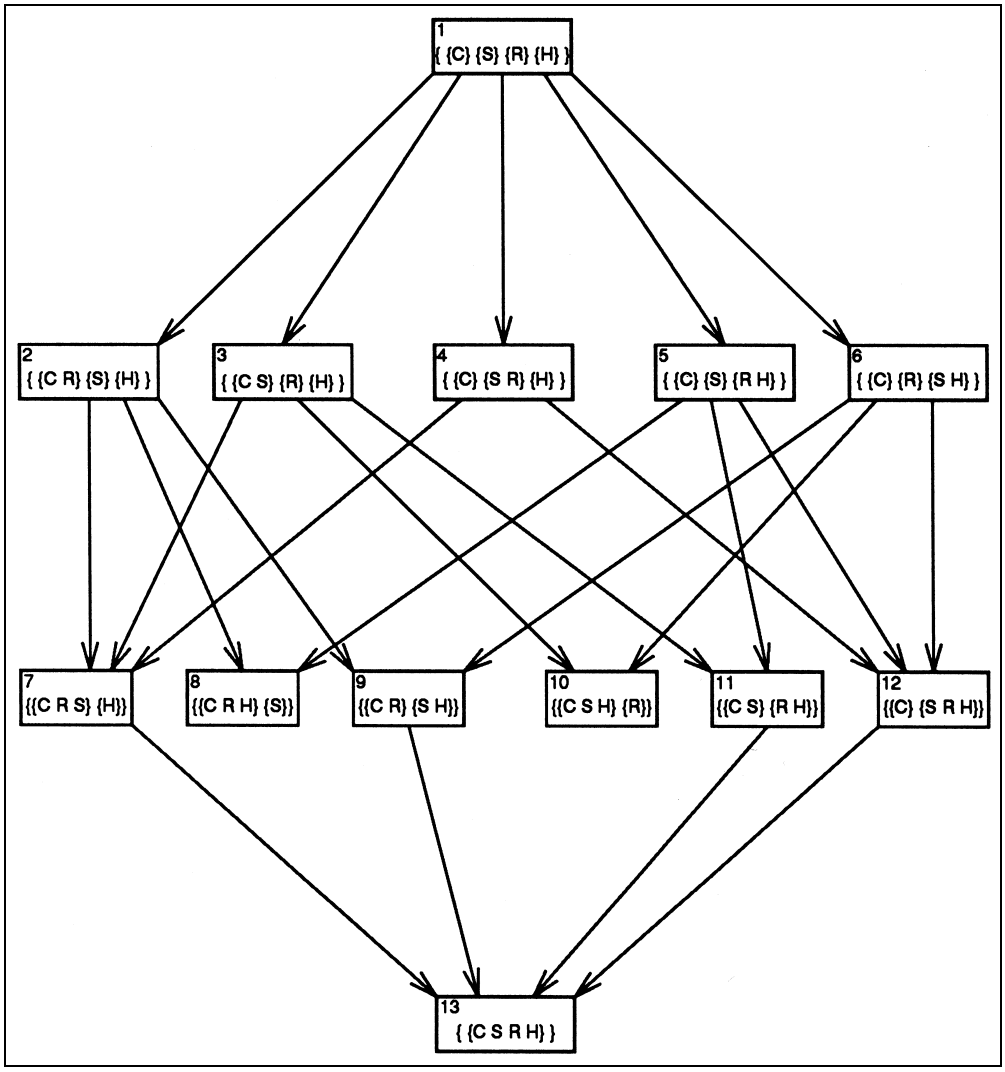


Abbildung 2.4: *Direkter Graph einer Montagesequenz mit den Bauteilen C,S,R,H (aus [RW91]).*

geometrischen und mechanisch ausführbaren Montageoperation. Jede Hyperkante ist ein geordnetes Paar, dessen erstes Element ein Knoten ist, der mit einem stabilen Teilaggregat θ_k korrespondiert, und dessen zweites Element eine Menge von zwei Knoten $\{\theta_i, \theta_j\}$ ist, so dass $\theta_i \cup \theta_j = \theta_k$. Existieren mehrere abgehende Hyperkanten an einem Knoten, so bezeichnen diese Alternativen in der Montage [Mos00].

Definition 2.2.2 *Ein* AND/OR Graph *einer durchführbaren Montagesequenz eines Aggregates, welches aus den Bauteilen der Menge* $P = \{p_1, \ldots, p_N\}$ *besteht, ist ein AND/OR Graph* $< S_P, D_P >$ *in dem*

$$S_P = \{\Theta \in \Pi(P) | \mathsf{sa}(\theta) \cap \mathsf{st}(\theta)\}$$

die Menge der stabilen Teilaggregate und

$$D_p = \{(\theta_k, \{\theta_i, \theta_j\}) | [\theta_i, \theta_j, \theta_k \in S_P] \wedge [\mathcal{U}(\{\theta_i, \theta_j\}) = \theta_k] \wedge [\mathsf{mf}(\{\theta_i, \theta_j\})] \wedge [\mathsf{gf}(\{\theta_i, \theta_j\})]\}$$

die Menge der durchführbaren Montageoperationen ist.

$\Pi(P)$ bezeichnet alle Mengen der Teilmenge von P.

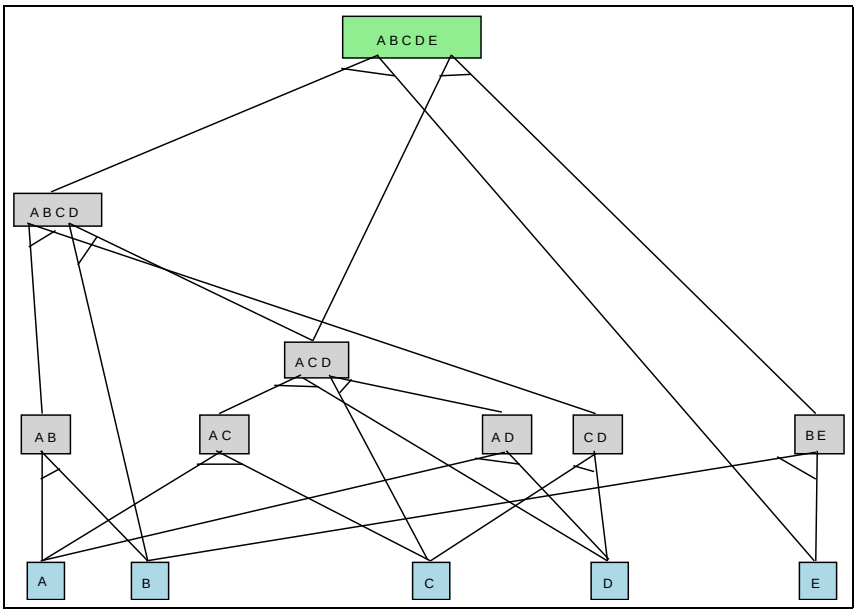


Abbildung 2.5: *AND/OR Graph für die Produktion von ABCDE.*

In [CS98] wird eine Erweiterung des AND/OR Graphen vorgestellt, das AND/OR-Netz. Ein AND/OR-Netz repräsentiert allgemeine geometrische Relationen und Randbedingungen von Objekten und Geräten in einem Robotersystem. Ist eine komplette geometrische Beschreibung von Objekten und Objektrelationen vorhanden, so liefert das AND/OR-Netz einen allgemeineren Zusammenhang zwischen Objekten als Systemunterzustände. Ein AND/OR-Netz ist ein Tripel (S, A, N). Die Elemente von S sind die Knoten, die Elemente von A sind die AND-Kanten und die Elemente aus N sind IST-Kanten (*Internal State Transition*). Im Gegensatz zu einem AND/OR Graphen ist ein AND/OR-Netz ungerichtet. Eine IST-Kante beschreibt eine Zustandsänderung des Systems, die nicht durch eine Montage- oder Demontageoperation hervorgerufen wurde. Damit beinhaltet ein AND/OR- Netz drei verschiedene Arten von Operationen: Montage- und Demontageoperationen, die durch AND-Kanten modelliert werden und interne Zustandsänderungsoperationen, die durch IST-Kanten modelliert werden.

Repräsentation von Verbindungsbedingungen

Werden die Zustände eines Montageprozesses als L-dimensionale binäre Vektoren dargestellt, dann kann eine Menge von logischen Ausdrücken verwendet werden, um eine direkte Graphdarstellung umzuwandeln [RW91]. Sei $\Xi_i = \{\vec{x}_1, \ldots, \vec{x}_{K_i}\}$ eine Menge von Zuständen, von der aus die i-te Verbindung hergestellt werden kann, ohne die Montagebeendigung auszuschließen. Die Bedingung zur Herstellung der i-ten Verbindung ist

dann eine logische Funktion

$$F_i(\vec{x}) = F_i(x_1, \ldots, x_L) = \sum_{k=1}^{K} \prod_{l=1}^{L} \gamma_{kl}$$

wobei die Summe und das Produkt die logischen Operationen AND und OR sind. γ_{kl} ist entweder x_l, wenn die l-te Komponente von $\vec{x}_k$ wahr ist oder $\bar{x}_l$, wenn die l-te Komponente falsch ist. $F_i(\vec{x}_k)$ ist nur dann wahr, wenn $\vec{x}_k$ ein Element von Ξ_i ist. Oft ist es möglich, die Ausdrücke von F_i mittels boolscher Algebra zu vereinfachen. Eine Montagesequenz, die eine geordnete Sequenz von Zuständen $(\vec{x}_1, \ldots \vec{x}_N)$ und deren Repräsentation eine geordnete Sequenz von Verbindungsbedingungen $(\gamma_1, \ldots, \gamma_{N-1})$ ist, kann nur dann ausgeführt werden, wenn die i-te Verbindung in der k-ten Operation hergestellt wurde. D.h. $F_i(\vec{x}_k)$ ist wahr.

APN (Assembly Petri Net)

Eine andere häufig verwendete Form der Repräsentation ist das Petri-Netz (siehe Anhang D). Auf Grund seiner formalen Natur erlaubt es die Ableitung von verschiedenen Eigenschaften wie Lebendigkeit, Sicherheit, Umkehrbarkeit [CS91] und strukturellen Eigenschaften [ea93]. In [TNB96] wird eine Repräsentation eines Montageplans mit Hilfe eines Petri-Netzes vorgestellt. Eine Marke in einer Stelle zeigt an, dass ein Objekt[1] repräsentiert durch die Stelle im aktuellen Stadium der Montage existiert. Das Petri-Netz muss die Montage- und Demontage eines Produktes repräsentieren, um Fehlerbehebung, Störungsbeseitigung und Reperatursequenzen zu modellieren.

Auf der Ebene des Montageplans existieren keine Referenzen über Montagehilfsmittel wie Roboter oder Puffer. Im Anfangsstadium der Montage sind alle Stellen, die unverbundene Bauteile repräsentieren, mit Marken belegt. Es existieren zwei Mengen von Stellen P_{IP} und P_{OP}, die das Anfangs- und Endstadium der Montage repräsentieren. Der Montageplan erhält von außen eine Menge von unverbundenen Bauteilen, die montiert werden sollen (Stellen P_{IP}), und die Ausgabe gibt das fertige Produkt nach außen ab (Stellen P_{OP}). Nach Vollendung eines Montageplans muss das Petri-Netz eines Montageplans in seinen initialen Zustand zurückkehren, so dass eine weitere Montage durchgeführt werden kann. In [TNB96] wird hierzu eine spezielle Transition t_{reset} eingeführt, um diesen Schritt explizit zu modellieren. Diese Transition hat die Stellen P_{IP} als Ausgabestellen und P_{OP} als Eingabestelle (Abb. 2.6).

Da die Montageplanung auch mit Echtzeitaspekten, wie z.B. der asynchronen Eigenschaft von Montage oder dem Zeitbedarf eines Montageschrittes, umgehen muss, wird das Netz in [TNB96] erweitert. Um den Zeitaspekt zu berücksichtigen, wird an jeder Transition vermerkt, wie lange diese Operation andauert. Da keine universale Zeit existiert, die die einzelnen Montageoperationen oder auftretende Ereignisse, wie das Eintreffen eines Bauteils, synchronisiert, bekommt jede Marke einen Zeitstempel. So

[1] Ein Objekt bezeichnet hierbei ein Bauteil oder eine Unterbaugruppe.

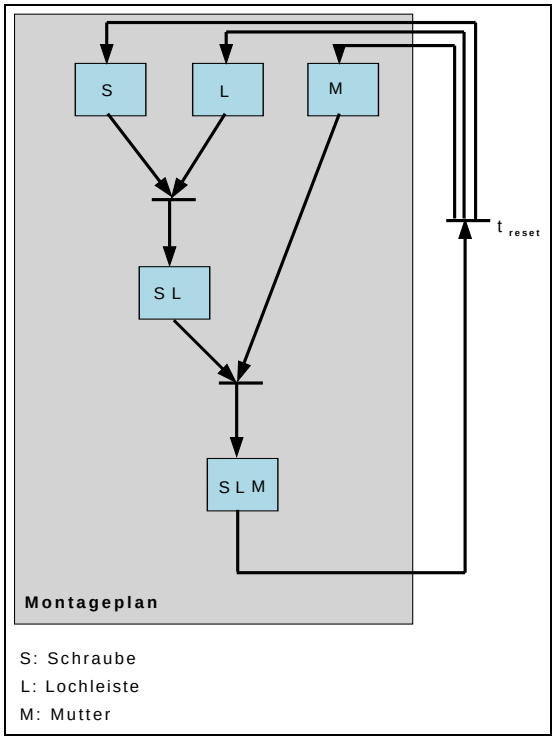


Abbildung 2.6: *Beispiel für einen Montagegraphen in Form eines Petri-Netzes.*

bezeichnen die Zeitstempel der Marken an den Stellen P_{IP} die Zeit des Eintreffens der initialen Bauteile.

Eine Transition schaltet, wenn an jeder Eingangsstelle der Transition eine Marke existiert. Der späteste Zeitstempel dieses Tupel von Marken kann als Schaltzeit der Transition bezeichnet werden. Während der Dauer, die die Transition schaltet, sind die Marken der Eingabestellen nicht mehr als Eingaben für andere Transitionen verfügbar, verbleiben aber an ihrem Platz bis die Operation, die durch die Transition repräsentiert wird, beendet ist. Ein Ergebnis dieser Operation ist das Entfernen dieses Tupels und die Erzeugung einer Marke in allen Ausgabestellen der Transition. Alle diese Marken haben denselben Zeitstempel, der den Endzeitpunkt der Operation bezeichnet.

Dadurch, dass ein APN formal ein Petri-Netz ist, hat das APN folgende Eigenschaften:

1. Sei das Petri-Netz N ein Montageplan, dann ist das Teilnetz N', welches ein Plan (unter vielen) ist, konfliktfrei und andauernd.
2. Ein Montageplan, der sowohl Montage- als auch Demontageoperationen enthält, so dass alle Montageoperationen reversibel sind, ist lebendig und sicher.
3. Das Netz terminiert nur, wenn das Produkt komplett montiert worden ist.
4. Das Netz kann nicht terminieren, wenn es nicht komplett initialisiert worden ist (es fehlt z.B. ein Bauteil).

Die *Lebendigkeit* des Netzes garantiert, dass der Plan keine *Verklemmung* besitzt und die Sicherheit garantiert, dass alle Objekte im Montageprozess nicht mehrfach vorkommen. Ausgehend von einer solchen Repräsentation kann man dann einen Plan zur Durchführung der Montage erstellen [TNB96].

Andere Repräsentationsformen

In [BKS99] wird ein hierarchischer Ansatz gewählt, um einen Montageplan zu repräsentieren und daraus eine Montagesequenz zu generieren. Ausgangspunkt ist die Unterteilung des zu montierenden Produktes in Bauteile mit *gebenden* Verbindungen (z.B Schrauben), nehmenden Verbindungen (z.B. Muttern) und Bauteilen, die durch Verbindungen von gebenden und nehmenden Bauteilen arretiert werden (z.B. eine Lochleiste). Da ein Objekt, das aus diesen drei Kategorien von Bauteilen zusammengesetzt worden ist, wiederum einer dieser Kategorien angehört, ergibt sich ein Graph mit einer hierarchischen Struktur, der das zu bauende Objekt über die Art und Weise, wie die Teile zusammengefügt werden, beschreibt. Durch das *bottom-up* Abschreiten des Graphen, kann ein Montageplan erstellt werden. Die Schwierigkeit dieses Ansatzes liegt in der Mannigfaltigkeit der möglichen Repräsentationen ein und desselben Endproduktes. Um eine durchführbare Montagesequenz für ein Produkt zu bekommen, müssen womöglich sehr viele Repräsentationen aufgestellt und getestet werden.

2.2.3 Weiterverarbeitung auf Montagesequenzen

Werden Montagegraphen verwendet, liegt das zu montierende Aggregat meist als Datenstruktur vor, die über ein CAD-Programm erstellt worden ist. Der Montagegraph entsteht dann durch die Demontage des Aggregates. Liegt ein Montagegraph vor, so lassen sich weitere Verarbeitungsschritte auf diesem Graph durchführen.

Ziel aller Schritte ist die Aufstellung einer Montagesequenz für die vorliegende Aktorik. In [Mos00] ist ein solches System vorgestellt. Dort wird anhand einer als CAD-Modell vorliegende Baugruppe die entsprechenden Montageoperationen generiert und anschließend in Elementaroperationen für einen Manipulator aufgespalten. Eine wichtige Voraussetzung ist die Umkehrbarkeit der Montageoperationen. Eine Baugruppe muss sich in umgekehrter Reihenfolge montieren lassen, wie sie demontiert werden kann. Für die Generierung der Montagesequenz existieren mehrere Randbedingungen, die bei der Planung berücksichtigt werden müssen [JW96]. Einige wichtige Kriterien sind z.B.:

- Notwendige Anforderungen:
  - Jedes Teilaggregat ist verbunden [AAB+91].
  - Die Montagesequenz beginnt mit einem existierenden Bauteil.
  - Ein Teilaggregat ist während einer Aktion (Transport, Montageoperation) stabil.

  - Ein auftretender Zustand ist im Plan enthalten.

- Optimierungen:
  - Maximierung eines Stabilitätsmaßes [MRW98, RMW97].
  - Minimierung von kritischen Operationen.
  - Minimierung der Montagekosten [MRW97].
  - Minimierung der Anzahl von Verbindungen, die gleichzeitig hergestellt werden [AAB+91].

- Weitere Bedingungen:
  - Ein Bauteil wird so früh wie möglich montiert.
  - Vorgeben einer Ordnung für die Montagereihenfolge.

2.3 Roboterprogrammierung über Instruktionen

Die Programmierung eines Roboters durch nicht explizite Angabe der spezifischen Roboterbefehle ist seit längerer Zeit Gegenstand der Forschung. Hierbei wird durch den Instrukteur eine Montagehandlung durchgeführt, welche vom Montagesystem verfolgt, aufgezeichnet und gegebenenfalls verallgemeinert wird. Von den verschiedenen Ansätzen einer solchen impliziten Programmierung werden zwei häufiger verfolgt. Der eine Ansatz besteht darin, über entsprechende Eingabehilfsmittel (z.B. Joystick, Datenhandschuh) direkt den Manipulator zu bewegen und mit dem Manipulator die Montagetätigkeit durchzuführen [WS98]. Es lassen sich hierbei sehr einfach die Sensordaten ermitteln und aufzeichen, so dass das Problem umgangen wird, Bewegungen des Instrukteurs auf die unterschiedliche Kinematik des Manipulators abzubilden. Der zweite Ansatz besteht darin, die direkt vom Instrukteur durchgeführten Handlungen zu detektieren und in entsprechende Bewegungen des Manipulators umzusetzen. Je nach Ausführung des Systems wird anschließend versucht, die durch den Instrukteur vorgegebenen Trajektorie mehr oder weniger genau zu kopieren bzw. zu optimieren [TTO+99, FHD98, WS98]. Eine weitere Methode ist das Versetzen des Roboters in den *zero-force* Modus und das direkte Führen des Manipulators per Hand [Mye99, AI89, FNUH01].

Auf diesen Ansätzen baut ist die Verallgemeinerung der gelernten Montagehandlung auf. Hierbei wird nicht mehr angenommen, dass die demonstrierte Trajektorie in jedem Fall optimal ist, sondern es wird versucht, diese auf elementare Operationen abzubilden.

In [FHD98] ist ein entsprechender Ansatz vorgestellt. Es werden für das später zu generierende Programm folgende Kriterien aufgestellt:

- Das Programm muss für den Benutzer interpretierbar sein.
- Aktionen müssen präzise beschrieben, aber auch flexibel gegenüber variablen Objektpositionen während der Ausführungszeit sein.

- Die Objektauswahl muss unterschiedlichen räumlichen Gegebenheiten angepasst werden.

Diese Ziele sollen erreicht werden, indem die Objektauswahl anhand von logischen Termen erfolgt. Es werden semantische Begriffe wie *in, auf, ausgerichtet, Farbe, Lot*, u.s.w. verwendet, die bei Bedarf entsprechend leicht angepasst bzw. modifiziert werden können. Die eigentlichen Bewegungen werden nicht anhand von globalen Koordinaten gespeichert, sondern werden relativ zu einem bestimmten, nicht fixen, Koordinatensystem aufgezeichnet, welches dann den Anforderungen entsprechend verschoben wird (z.B. beim Greifen). Objekte, die durch die Objektauswahl ausgewählt worden sind, können bei Bedarf mit einer externen geometrischen Beschreibung abgeglichen und nachträglich verworfen werden. Toleranzen werden berücksichtigt. Die angestrebte Flexibilität soll zusätzlich durch das Abbilden der aufgenommenen Trajektorie auf Elementaroperationen erreicht werden. Diese Elementaroperationen können nachträglich vom Instrukteur optimiert werden.

Ein ähnlicher Ansatz wird in [BA01] wird verfolgt. Dort wird einem Roboter das Spielen von *Air Hockey* beigebracht. Das System beobachten den Instrukteur, den Puck, dessen Geschwindigkeiten und ordnet die beobachteten Bewegungen bestimmten vorgegebenen Bewegungsprimitiven zu. Der Ansatz ist erfolgreich in einer Simulationsumgebung getestet worden.

In [CM98, TS99] wird eine Bewegung als Abfolge von Ereignissen betrachtet, die während der Bewegungsdemonstration auftreten. Ein solches Ereignis ist z.B. ein eindeutiger Kontaktzustand zwischen dem Endeffektor und seiner Umgebung. Ziel ist es, später diese Zustände nacheinander einzunehmen und dadurch eine entsprechende Bewegung auszuführen, die die gewünschte Operation durchführt. Durch die Betrachtung von Kontaktzuständen wird die Montagehandlung positions- und orientierungsunabhängig. Der Ansatz wird in [CM00] erweitert.

In [MPB01] werden die Bewegungen eines Manipulators durch die Geschwindigkeitsvektoren repräsentiert und die Daten eines Kraftmomentensensors genutzt, um den Übergang von einer Bewegung zur anderen zu detektieren.

Soweit bekannt beschäftigen sich alle Ansätze mit dem Erlernen einer oder mehrerer Montageoperationen (z.B. [Kai96]) und nicht mit der Erlernung der Montage eines komplexeren Bauteils. Da viele Ansätze mit Hilfe der Daten der Kraftmomentensensoren ihre Zustandsübergänge detektieren, ist ein kontaktloser Zustand, wie er bei der Aggregatmontage oft vorkommt, nicht vorgesehen. Es wird hauptsächlich versucht, eine Invarianz des Systems gegebenüber Bauteilpositionen und -orientierungen zu erreichen bzw. die gelernte Trajektorie nach einem Maß zu optimieren. Mehrrobotersysteme werden hierbei nicht berücksichtigt.

In der vorliegenden Arbeit ist die Abstraktionstufe höher als in den sonst üblichen Arbeiten über *Programming by Demonstration*. Es wird davon ausgegangen, dass elementare Montageoperationen, wie die z.B. in [Kai96] akquiriet werden, zur Verfügung stehen und es soll die Montage einer komplexen Baugruppe erlernt werden.

Kapitel 3

Integrierte Robotersteuerung

3.1 Übersicht

Das in dieser Arbeit entwickelte *OPERA* ist eine Entwicklungs- und Arbeitsumgebung für robotikspezifische Montageanwendungen und unterstützt deren Erstellung durch die Bereitstellung einer einheitlichen Schnittstelle zwischen den einzelnen Teilkomponenten. Da *OPERA* nicht als in sich geschlossenes System konzipiert ist, sondern auf unterschiedlichsten Komplexitätstufen erweitert und ausgebaut werden kann, ist nicht nur die Anwendung in der direkten Roboterprogrammierung möglich, sondern auch die Einbeziehung anderer benachbarter Teilgebiete wie z.B. der Bildverarbeitung. Das System hat folgende Merkmale:

- Aufgabenorientierte Programmentwicklung mit sensorbasierten Operationen.
- Eine Kombination aus Entwicklungs-, Test- und Ablaufumgebung mit einer graphischen Benutzerschnittstelle für kurze Entwicklungszyklen und Sensorüberwachung.
- Ein objektorientierter Ansatz für einfache Wiederverwendung von Quelltexten und Operationen.
- Eine einfache und klare Programmschnittstelle mit geringen Restriktionen und der Möglichkeit der Änderung oder Erweiterung von Operationen und ihrer Parameter. Diese beinhaltet eine einheitliche und abstrakte Schnittstelle zur Roboter- und Sensoransteuerung.
- Integrierter Skriptinterpreter.

Bei der Konzeption und Realisierung des Systems wurde vor allem darauf Wert gelegt, dass das System die benötigten Mechanismen zur Repräsentation und Generalisierung von Montageabläufen bereitstellt. So wurde unter anderem ein eigener Skriptinterpreter implementiert, da die vorhandenen Sprachen die benötigten Strukturen und Funktionalitäten nicht bereitstellen oder nur sehr aufwendig nachgebilden können. Gleich-

zeitig fehlen dem in *OPERA* integriertem Skriptinterpreter Konstrukte (Wertzuweisungen, arithmetische Ausdrücke, Schleifen) , die in anderen Sprachen vorhanden sind. In [Smi92] werden beispielsweise C-Programme zur Ablaufsteuerung verwendet, die von einem C-Interpreter verarbeitet werden.

3.2 Definitionen

Für die Beschreibung der Funktionalität und Semantik des *OPERA*-Systems, insbesondere des Interpreters, werden zunächst einige Definitionen eingeführt, auf die im späteren Verlauf der Arbeit zurückgegriffen wird. Sei:

$\mathbb{G}$**:** Die Menge der reellen Zahlen.

$\mathbb{N}$**:** Die Menge der natürlichen Zahlen.

$\mathbb{Z}$**:** Die Menge der ganzen Zahlen.

Σ**:** Die Menge aller Instanzen von Klassen, die von der Klasse `TOCCLObject`[1] abgeleitet sind:

$$\Sigma \stackrel{\text{def}}{=} \{x \mid x \text{ ist (abgeleitete) Instanz von } \texttt{TOCCLObject}\}$$

Eine weitergehende Definition der Menge Σ ist an dieser Stelle nicht von Interesse, da der Aufbau der Elemente dem Gesamtsystem im Allgemeinen nicht bekannt ist.

T: Die Menge aller nullterminierten Zeichenketten.

Ist a ein n-Tupel, so bezeichnet a_i das i-te Element von a mit $1 \leq i \leq n$. Des Weiteren ist:

Zustand eines Roboter:

$$r \stackrel{\text{def}}{=} \{(\Theta_1, \ldots, \Theta_6, g) \mid \Theta_i \in \mathbb{G};\ i = 1, \ldots, 6;\ g \in \{0, 1\}\}$$

mit Θ_i als Gelenkwinkel des i-ten Gelenkes. r beinhaltet alle möglichen Gelenkkonfigurationen $(\Theta_1, \ldots, \Theta_6)$ und Greiferkonfigurationen (g) eines Roboters[2] und damit beschreibt r_i die Menge aller möglichen Konfigurationen des Roboters i. Damit ist

$$R \stackrel{\text{def}}{=} r^n \qquad n \in \mathbb{N}$$

die Menge alle Zustände von n Robotern.

[1] Siehe Anhang: OCCL.

[2] Da alle praktischen Versuche mit einem sechsgelenkigen Roboter mit einem pneumatischen Zweibackengreifer gemacht wurden, wird an dieser Stelle von dieser Hardwarekonfiguration ausgegangen. Dies ist aber keine allgemeine Einschränkung.

Blackboard

$$B(\tau) \stackrel{\text{def}}{=} \{((t_1, x_1), \ldots, (t_{n(\tau)}, x_{n(\tau)}))\}$$

mit $x_i \in \Sigma\ ;\ t_i \in T;\ i = 1, \ldots, n; n(\tau) = \mathbb{Z}_0^+$. Ein *Blackboard* sei ein globaler Speicherpool in dem Elemente aus Σ unter einem Namen hinterlegt werden können. Da zu unterschiedlichen Zeitpunkten τ eine unterschiedliche Anzahl von Elementen abgespeichert ist, ist die Größe des n-Tupels zeitveränderlich und hängt damit von τ ab.

Sensorzustand: $S \stackrel{\text{def}}{=} \mathbb{G}^n$ mit $n \in \mathbb{N}$

Der Zustand des Gesamtsystems ist damit definiert als:

$$G \stackrel{\text{def}}{=} \{(\rho, \beta, s) \mid \rho \in R, \beta \in B, s \in S\}$$

3.3 Systemaufbau

Das Basissystem von *OPERA* besteht aus vier Kernkomponenten:

- Benutzerschnittstelle,
- Skriptinterpreter (oder auch Sequenzinterpreter),
- *Blackboard* und
- Modulmanagement inkl. Verwaltung der Roboter.

Die Komponente (Abb. 3.1) für die Benutzerschnittstelle regelt die gesamte Kommunikation mit dem Benutzer und führt sämtliche Ein- und Ausgabeoperationen vom und zum Benutzer durch. Der Skriptinterpreter interpretiert die über *OPERA* erstellten Skripte. Das *Blackboard* dient dem Gesamtsystem als globaler Speicherpool, in dem alle Komponenten des Systems Daten unter einem eindeutigen Namen ablegen können. Es dient weniger zur Kommunikation der Komponenten des zentralen Management als vielmehr dem Datenaustausch zwischen verschiedenen nachträglich zum System hinzukommenden Komponenten, die ihre interne Struktur gegenseitig nicht unbedingt kennen. Da *OPERA* selbst keine Funktionalität zur Ansteuerung der Roboter und zur Montage beinhaltet, wird das System über sog. Module erweitert, die diese Funktionalität zur Verfügung stellen. Die gesamte Verwaltung dieser Module wird durch das Modulmanagement realisiert. Die Robotersteuerung wird ebenfalls durch ein Modul realisiert und deren Verwaltung wird daher auch durch das Modulemanagement übernommen.

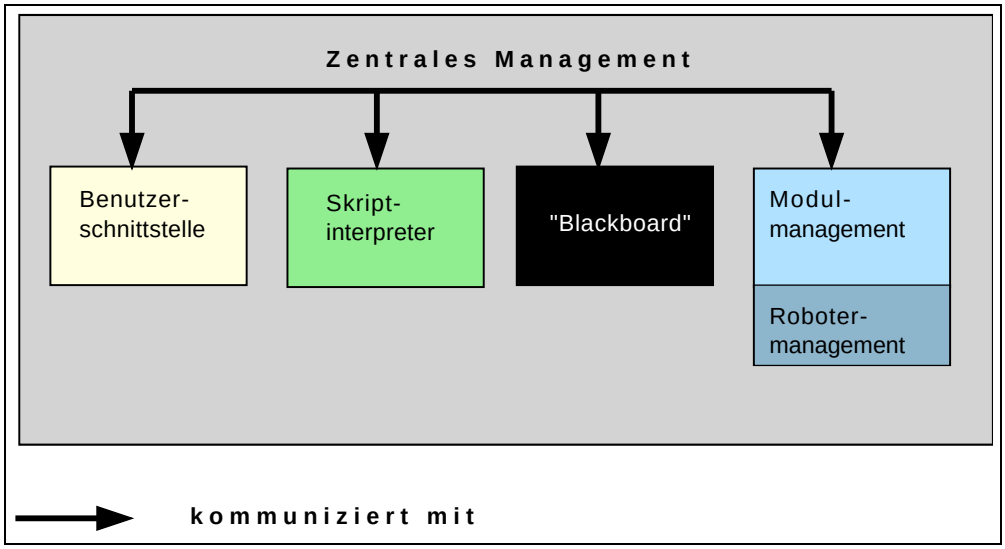


Abbildung 3.1: *Schema des Gesamtsystems. Das Kernsystem besteht nur aus der Benutzerschnittstelle, einem Skriptinterpreter, dem* Blackboard *und dem Modulmanagement. Alle weiteren Funktionalitäten müssen über Module realisiert werden.*

3.3.1 Module / Modulemanagement

Über Module wird *OPERA* in sehr verschiedener Weise erweitert, wobei eine objektorientierte Sichtweise verwendet wird. Jedes Modul repräsentiert eine Aktion oder Operation:

$$P \stackrel{\text{def}}{=} \{(p_1, \ldots, p_m) \mid p_i \in \Sigma;\ m \in \mathbb{Z}\} \tag{3.1}$$

$$M : (G, P) \rightarrow G \tag{3.2}$$

P bezeichnet die Menge aller möglichen Parameter und M_i ist eine Zustandstransformationsfunktion des Moduls i mit $i \in T$. Wird ein Modul zu *OPERA hinzugeladen*, wird das Gesamtsystem um diese zusätzliche Funktionalität erweitert und es kann auf alle übrigen Komponenten des System zugreifen (Abb. 3.2). Auch alle anderen Module können sich sofort dieser neuen Funktionalität bedienen. Die in ihm definierte Operation steht über eine parametrisierte Methode zur Verfügung. Module können ihrerseits Module zum System hinzufügen und sich deren Funktionalität bedienen. Über die einheitliche Methodenschnittstelle wird erreicht, dass die unterschiedlichsten Module untereinander kommunizieren bzw. sich *aufrufen* können, ohne direkte Kenntnisse von anderen Modulen zu besitzen (Abb. 3.3). Dabei ist die Aktion, die in einem Modul implementiert ist, nicht vordefiniert und nicht auf die Robotik beschränkt. Dies können Berechnungen verschiedener Art oder Operationen zur Steuerung von Manipulatoren und zur Sensorauswertung sein. Zusätzlich können durch Module zentrale Funktionen von *OPERA* erweitern werden. So ist das im zweiten Teil dieser Arbeit vorgestellte Lernverfahren nicht direkt im Basissystem von *OPERA* integriert, sondern in Form von Modulen realisiert und bedient sich nur der durch das System bereitgestellten Kernfunktionalität.

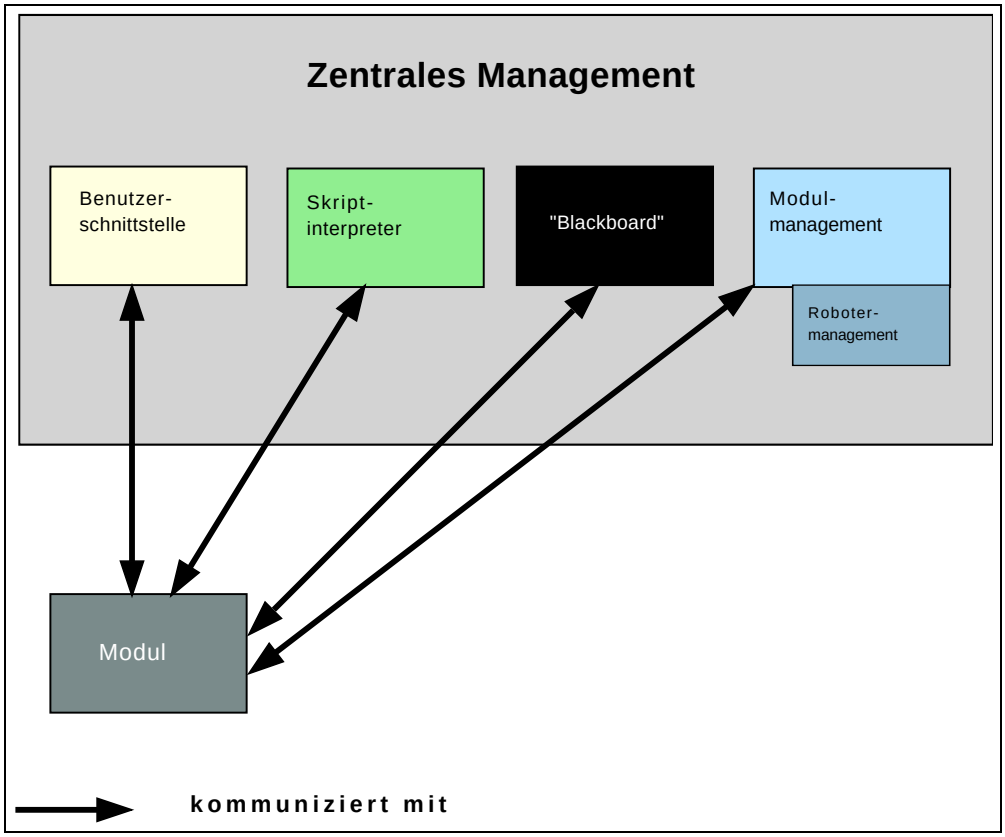


Abbildung 3.2: *Kommunikationsverbindungen eines Moduls.*

Die Kombination aus einem Modul und einem Parametersatz für die im Modul definierte Operation wird hier als Kommando bezeichnet. Sei C_M die Menge aller möglichen Kommandos:

$$C_M \stackrel{\text{def}}{=} \{(m,p) \mid m \in \{M : (G,P) \rightarrow G\}, p \in P\}$$

Eine Sequenz von solchen Kommandos bildet dann ein Skript, die eigentliche Applikation. Skripte sind ebenfalls parametrisierbar und es besteht die Möglichkeit in ihnen, zusätzlich zu Kommandoaufrufen, einfache Verzweigungen und Aufrufe weiterer Skripte zu generieren.

Kategorien

Ein Modul kann einer oder mehrerer Kategorien angehören[3]:

`INTERACTIVE:` Das Modul bietet dem Benutzer eine Schnittstelle an, die Parameter für seinen Aufruf interaktiv zu spezifizieren.

`RUN:` Das Modul wird sofort nach dem Hinzufügen ins System aktiviert.

[3]Siehe auch Anhang: Programmierung eines *OPERA*-Moduls.

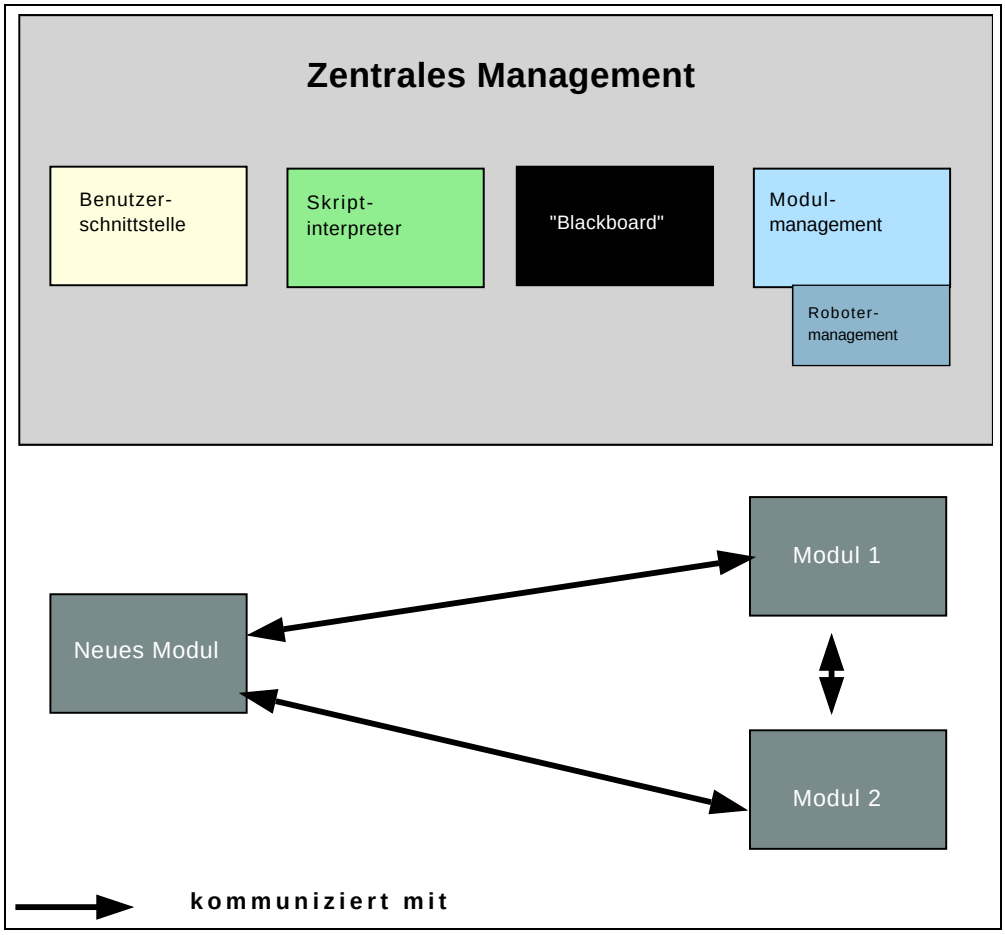


Abbildung 3.3: *Interkommunikationsmöglichkeiten von Modulen.*

EVENT: Dieses Modul bearbeitet Ausnahmezustände des Systems.

SENSOR: Das Modul stellt die Daten eines Sensors zur Verfügung.

Je nachdem welcher Kategorie ein Modul angehört, stellt es unterschiedliche Funktionalitäten zur Verfügung und kann vom Restsystem verschieden angesprochen werden.

Die Philosophie besteht darin, eine Applikation hierarchisch aufzubauen. Operationen auf einer niedrigen Abstraktionsebene werden als Module in der Programmiersprache C++ geschrieben. Operationen auf höhereren Abstraktionsebenen können als Modul in C++ oder als Kommandosequenz implementiert werden. Dadurch werden eine hohe Wiederverwendbarkeit von unterschiedlichen Modulen und variable Erweiterungsmöglichkeiten auf unterschiedlichen Komplexitätsebenen erreicht.

Modulmanagement

Das Modulmanagement verwaltet im System die Module. Es ist zuständig für das Hinzu- und Herausnehmen von Modulen ins und aus dem System und für die Verwaltung von

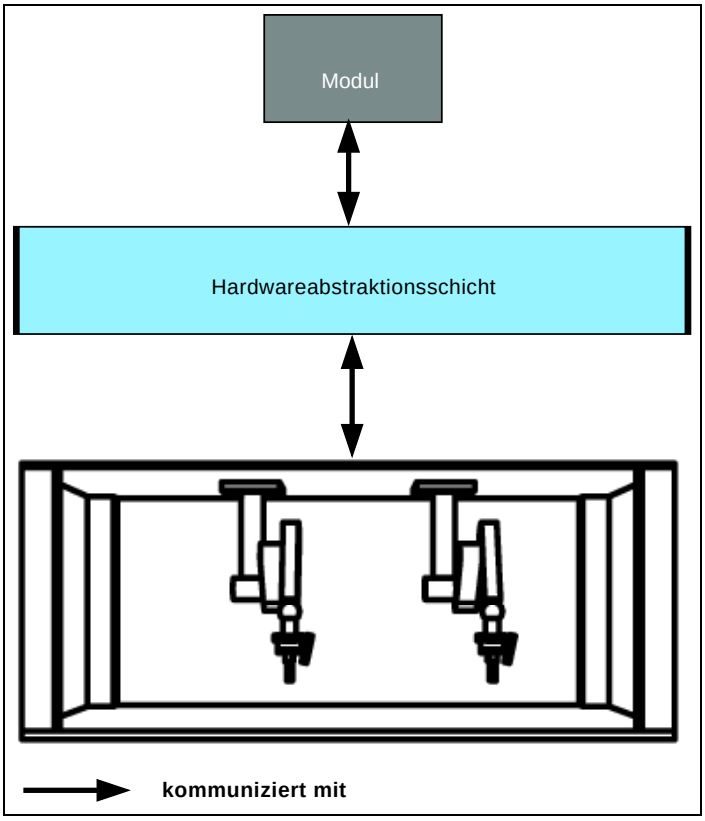


Abbildung 3.4: *Schnittstelle zu einem realen Roboter. Ein Modul kommuniziert nicht direkt mit der Robotersteuerungssoftware, sondern bedient sich einer Zwischenschicht, die die Anweisungen an die Steuersoftware weiterreicht.*

modulspezifischen Informationen. Zusätzlich dient es allen Komponenten als Informationsquelle über modulspezifische Daten und ist für die korrekte Initialisierung und Deinitialisierung neuer Module zuständig.

Roboteransteuerung

Mit *OPERA* kann eine beliebige Anzahl von Robotern und unterschiedlicher Robotertypen gesteuert werden. Um nicht für jeden neuen Robotertyp einen neuen Satz von Funktionen zur Ansteuerung zu erhalten, werden die Roboter nicht direkt über eine relativ hardwarenahe Funktionsbibliothek (z.B. RCCL [HP86]) angesprochen, sondern über eine Hardwareabstraktionsschicht, die von den einzelnen Robotertypen abstrahiert (Abb. 3.4). In dieser Zwischenschicht ist ein fester Satz von Methoden definiert, welche entsprechend auf die für den jeweiligen Roboter existierende Steuerung abgebildet werden. Diese Methoden reichen von einfachen kartesisch- oder gelenkinterpolierten bis hin zu komplexen nachgiebigen und geregelten Bewegungen (siehe Anhang A.2.1). Ein Modul, welches die Roboter steuern möchte, erhält die dafür notwendigen Informatio-

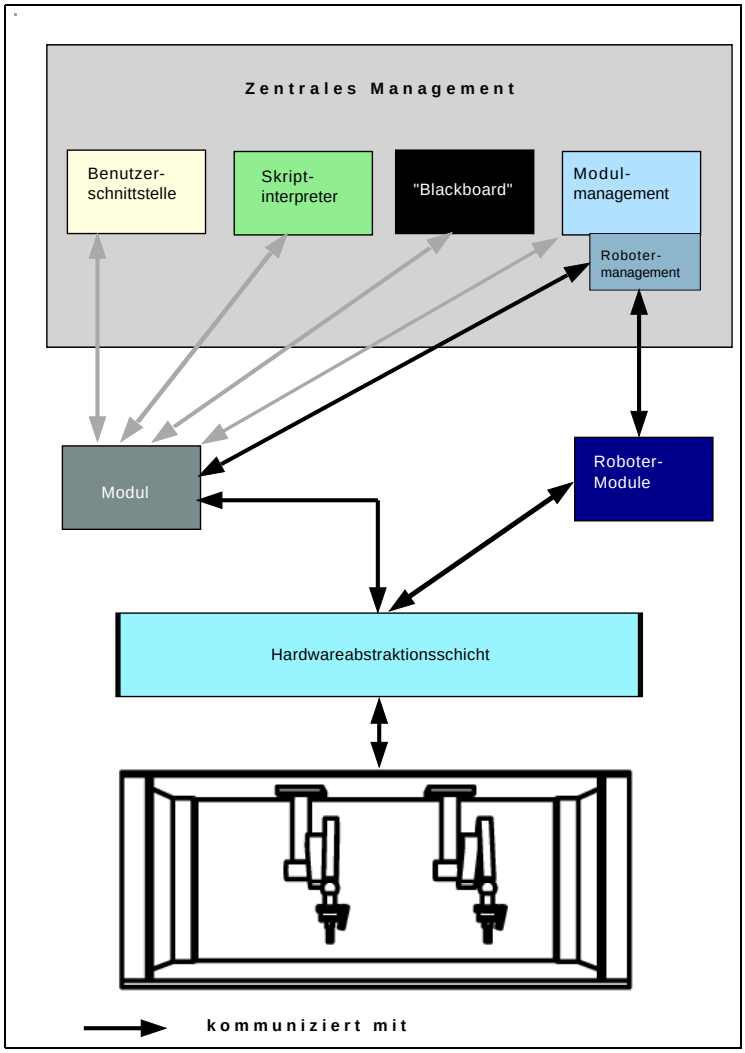


Abbildung 3.5: *Kommunikationsverbindungen eines Moduls zum Roboter. Ein Modul bekommt über das Robotermanagment die notwendigen Informationen zur Steuerung der Roboter.*

nen vom Robotermanagement (Abb. 3.5), das seinerseits die alle für sich notwendigen Informationen von einem Robotermodul erhält. Dieses Robotermodul ist ein spezielles Modul und verwaltet die von ihm zur Verfügung gestellten Roboter. Mit Hilfe der Informationen vom Robotermanagement ist ein Modul in der Lage, seine Kommandos direkt an die Hardwareabstraktionsschicht zu übergeben und die Roboter zu steuern. Die Abstraktionsschicht stellt sog. *virtuelle Roboter* zur Verfügung, welche von der Applikation gesteuert werden. Über diese Abstrahierung ist es möglich, transparent mit verschiedenen Robotertypen zu arbeiten und ist damit unabhängig von der jeweiligen Programmiersprache der Roboterhersteller. Die Anzahl der virtuellen und real vorhandenen Roboter kann differieren. Es kann z.B. auch ein Verbund von einzelnen Robotern zu einem *virtuellen* Roboter zusammengefasst werden, ohne dass sich die Ansteuerung

für ein Modul der Applikation ändert. Abb. 3.6 zeigt hierfür ein Beispiel. Während real vier Roboter vorhanden sind, stehen der Applikation nur zwei Roboter zur Verfügung. Die Komponente zur Verwaltung dieser *virtuellen* Roboter setzt die Anweisungen entsprechend auf die vier realen Roboter um. Dieses Schnittstellenkonzept ermöglicht

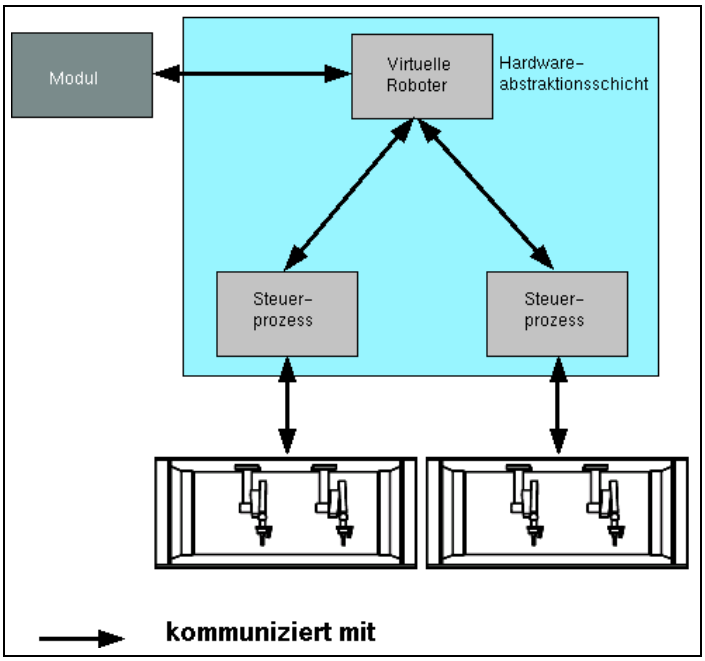


Abbildung 3.6: *Schnittstelle zu einem virtuellen Roboter. Die Informationen über die Anzahl der Roboter muss nicht unbedingt mit der realen Anzahl übereinstimmen. Ein Anwendung kann z.B. nur Informationen über zwei Roboter besitzen, obwohl die Steueranweisungen auf vier reale Roboter umgesetzt werden.*

es, die reale Hardware auf vielfältige Weise zu verändern, ohne das die Anwendung mit großem Aufwand reprogrammiert werden muss. Das System gewinnt an Flexibilität. Diese Schnittstellenkonzept ist auch für den Fall angewandt worden, dass die verwendeten Manipulatoren von unterschiedlichen Rechnern gesteuert werden (siehe Abb. 3.7). Es werden die Steuerkommandos von der Hardwareabstraktionsschicht entgegengenommen und an den jeweiligen Steuerrechner übertragen (z.B. über Ethernet).

Dieses Konzept der Roboteransteuerung die Schnittstelle auf einem hohen Abstraktionsniveau anzusetzen führt dazu, dass Operationen, die eine enge Regelschleife benötigen (z.B. Kraftmomentenregelung), in der Hardwareabstraktionsschicht implementiert werden müssen und daher diese Schicht zum Teil sehr komplexe Funktionen beinhaltet, wie z.B. Schraub- und Suchbewegungen, bei denen die Kräfte entlang bestimmter, aber nicht aller, Toolachsen geregelt werden. Die Implementierung von Regelalgorithmen in *OPERA* ist nicht vorgesehen. So existiert kein Mechanismus für Kontroll- oder Monitorfunktionen wie in entsprechenden Robotersteuerungssprachen bzw. -bibliotheken [HP86, Sch00b], die synchron zum Trajektoriengenerator laufen und Einfluss auf die

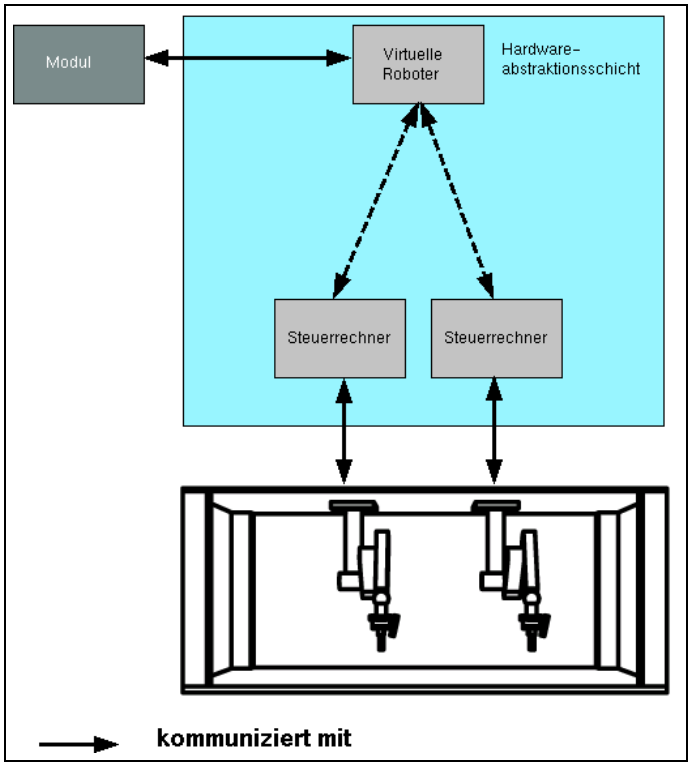


Abbildung 3.7: *Steuerung von Manipulatoren deren Steuersoftware auf mehrere Steuerrechner verteilt ist. Die Steueranweisungen an den Roboter werden von der Hardwareabstraktionsschicht nicht direkt an die Robotersteuerungssoftware übergeben. Da die Roboter von unterschiedlichen Rechnern kontrolliert werden, werden die Anweisungen erst an diese Rechner gesendet und dort von der Steuersoftware umgesetzt.*

Steuerung des Roboters nehmen. Solche Funktionen müssen in der Hardwareabstraktionsschicht realisiert werden. Aufgrund der von der Hardwareabstraktionsschicht zur Verfügung gestellten zum Teil sehr komplexen Handlungsprimitive muss die verwendete Software zur Ansteuerung der realen Hardware folgende Merkmale besitzen:

- Bereitstellung von kartesisch- und gelenkwinkelinterpolierten Bewegungen.
- Die Möglichkeit Kräfte und Drehmomente am Endeffektor während einer Bewegung entlang von unterschiedlichen Koordinatenachsen zu regeln und/oder zu überwachen.
- Das Erreichen des Zielpunktes bedeutet nicht zwangsläufig das Ende der Bewegung (unendliche Bewegungen).
- Die Nebenläufigkeit von Kontrollebene und Steuerungsebene (Trajektoriengenerator).

- Das Absätzen von sich zeitlich überschneidenden Bewegungen an zwei unterschiedliche Manipulatoren.
- Das Abbrechen einer Bewegung zu einem beliebigen Zeitpunkt.
- Berechnung der inversen Kinematik.

Diese Anforderungen werden z.B durch die C-Bibliothek *RCCL* [HP86] und nur teilweise durch die C++-Bibliothek *COMA* [Sch00b] und die Sprache *V+* [Pro93] erfüllt. *COMA* fehlen Mechanismen zur Einflussnahme auf eine Bewegung und in der Sprache *V+* werden Bewegungen bei Erreichen des Trajektorienendpunktes beendet. Diese Funktionalität wird z.B. benötigt, um mit dem Endeffektor bestimmte Andruckkräfte einzuregeln.

Eine andere Möglichkeit wäre die Implementierung einer abstrakten Schnittstelle zur Realisierung von engen Regelschleifen. Eine solche Schnittstelle müsste aber sehr allgemein gehalten werden, um auf unterschiedlichen Robotersteuerungen implementiert werden zu können. Es kann nicht abgesehen werden, ob eine solche Schnittstelle den Portierungsaufwand reduziert. Die Problematik der Nutzung einer fremden Steuerungssoftware kann durch die Integration einer portablen Echtzeit-Kontrollarchitektur und der Robotersteuerung in *OPERA* umgangen werden. Eine Ansatz für eine solche Kontrollarchitektur wird in [FBW01] vorgestellt. Dort übernimmt ein *Objekt Server* die Koordination zwischen den einzelnen Komponenten der Steuerungssoftware. Durch die Verwendung eines *Message Passing* Protokolls sind die einzelnen Komponenten modularisiert und können analog zu den Modulen in *OPERA* transparent ausgetauscht werden. Andere Ansätze sind in [TAG01, TS99, ea99, WRS99, WS01] zu finden. In [TAG01] werden Softwarekomponenten in funktionale Einheiten zusammengefasst (Algorithmen, Ansteuerung von Sensoren und Aktuatoren) und analog zur Schaltungsentwicklung in der Elektronik zusammengesetz und miteinander verschaltet. Eine Bibliothek von Software-Modulen zur Entwicklung von verteilten Echtzeit-Bewegungssteuerungen wird in [TS99] vorgestellt. Es hat den gleichen Client-Server Ansatz wie [FG98] berücksichtigt aber die Echtzeitbedingungen, die für die Steuerung Voraussetzung sind.

Ein Verlagerung der Schnittstelle zwischen der Anwendung und der realen Hardware auf eine hardwarenahe Schicht führt aber dazu, dass die Flexibilität bei der Verwendung unterschiedlicher Manipulatortypen nicht mehr gegeben ist. Eine Verwendung mehrerer Steuerrechner (Abb. 3.7) ist über eine solche Schnittstelle nicht mehr möglich.

3.3.2 Sequenzen / Skriptinterpreter

Wie oben erwähnt bildet ein Skript in *OPERA* die eigentliche Montageanwendung. Eine solches Skript wird durch das Erstellen einer Sequenz von Kommandos gebildet, welche vom Skriptinterpreter (SI) interpretiert wird. Im Folgenden wird die Instruktions- und Programmsemantik nach [Kla90] erläutert.

Der Zustand des SI ist gegeben durch den Zustand des Systems G und dem Instruktionszeiger z mit $z \in \mathbb{Z}^+$. Ein Zustand des SI ist damit ein Tupel (g, z) mit $g \in G$. Die

Zustandsmenge U ist damit definiert als:

$$U \stackrel{\text{def}}{=} G \times \mathbb{Z}^+$$

Die Menge C der SI-Operationen ist definiert durch

$$C \stackrel{\text{def}}{=} C_0 \cup C_1 \cup C_2 \cup C_{\mathsf{m}} \cup C_4$$

wobei

$$\begin{aligned} C_0 &= \{\mathsf{STOP}\} \\ C_1 &= \{\mathsf{JUMP}\} \\ C_2 &= \{\mathsf{IF}\} \\ C_{\mathsf{m}} &= \{\mathsf{DLL}\} \\ C_4 &= \{\mathsf{CALL}\} \end{aligned}$$

Die Menge I der SI-Instruktionen ist damit:

$$\begin{aligned} I \stackrel{\text{def}}{=} C_0 \quad &\cup \quad C_1 \times T && (3.3) \\ &\cup \quad C_2 \times T \times T && (3.4) \\ &\cup \quad C_{\mathsf{m}} \times P \equiv C_M && (3.5) \\ &\cup \quad C_4 \times \{(t, x_1, \ldots, x_n) \mid t \in T; x_{1,\ldots,n} \in \Sigma\} && (3.6) \end{aligned}$$

Instruktionssemantik

Die Semantik der $\mathcal{I}[i]$ einer SI-Instruktion $\mathcal{I} \in I$ ist eine Zustandstransformation; d.h. eine Abbildung $U \to U$. Die Semantikfunktion ist also eine Abbildung

$$\mathcal{I}[i] : I \to [U \to U]$$

Darüber hinaus sei:

$$L[t] : T \to \mathbb{Z}^+$$

eine Funktion, die einen Text in eine ganze Zahl umwandelt. Sie definiert den Zusammenhang zwischen einer Marke und der entsprechend mit dieser Marke markierten Instruktion. Die Funktion

$$\mathsf{Bool}[t, g] : T \times G \to \{0, 1\}$$

sei eine Funktion, die aus einem Text und dem Zustand des Gesamtsystems einen boolschen Wert ermittelt.

Damit sind die einzelnen Instruktionen des SI wie folgt definiert:

$$\begin{aligned}
\mathcal{I}[\text{STOP}](g,z) &\stackrel{\text{def}}{=} (g,z)\\
\mathcal{I}[\text{JUMP t}](g,z) &\stackrel{\text{def}}{=} (g,L(t))\\
\mathcal{I}[\text{IF j,t}](g,z) &\stackrel{\text{def}}{=} \begin{cases} (g,L(t)) & : \text{Bool}(j,g)\neq 0\\ (g,z+1) & : \text{Bool}(j,g)=0 \end{cases}\\
\mathcal{I}[\text{DLL i,p}](g,z) &\stackrel{\text{def}}{=} (M_i(g,p),z+1)\\
\mathcal{I}[\text{CALL n,k}](g,z) &\stackrel{\text{def}}{=} (\zeta_n(g,k),z+1)
\end{aligned}$$

Sei ζ_i eine Funktion $G \to G$ und beschreibt die Zustandstransformation durch das Interpretieren einer Sequenz ζ_i.

$$\zeta[g,k] : (G,P) \to G$$

Programmsemantik

Sei $\pi : \mathbb{N} \to I$ eine Sequenz. Die Einzelschrittfunktion Δ_π ist eine Zustandstransformation

$$\Delta_\pi : \quad U \to U$$

mit

$$\Delta_\pi(g,z) \stackrel{\text{def}}{=} \mathcal{I}[\pi(z)](g,z).$$

Sie beschreibt die Zustandsänderung, durch die ein Sequenzschritt bei der Ausführung von π bewirkt wird. Die Programmsemantik besteht dann aus einer Iteration der Einzelschrittfunktion $\Delta_\pi^\infty : U \to U$:

$$\Delta_\pi^\infty(g,z) \stackrel{\text{def}}{=} \begin{cases} (g,z) & \text{falls } \Delta_\pi(g,z)=(g,z)\\ \Delta_\pi^\infty(\Delta_\pi(g,z)) & \text{andernfalls} \end{cases}$$

Beispiel Sei nun beispielsweise:

$$\begin{aligned}
\mathcal{I}[M_1] &= \text{Initialisiere das System}\\
\mathcal{I}[M_2] &= \text{Greife, wobei die Parameter Roboter und das zu greifende Objekt angeben}\\
\mathcal{I}[M_3] &= \text{Fahre Roboter vor; Parameter ist der Roboter}\\
\mathcal{I}[M_4] &= \text{Suche Loch; Parameter ist der Roboter}\\
\mathcal{I}[M_5] &= \text{Stecke Schraube ins Loch; Parameter ist der Roboter}\\
\mathcal{I}[M_6] &= \text{Öffne Hand; Parameter ist der Roboter}\\
\mathcal{I}[S_1] &= \text{Fahre Roboter aufeinander zu}
\end{aligned}$$

$$L(\text{“Label1”}) = 8$$

Ein mit diesen Funktionen erstelltes Skript können dann zum Beispiel so dargestellt werden:

```
1  DLL ‘‘Init System’’
2  IF 1 > 2, ‘‘Label1’’
3  DLL ‘‘Grasp’’,0,(‘‘Screw’’,’’yellow’’)
4  DLL ‘‘Grasp’’,1,(‘‘Ledge’’)
5  JUMP 8
6  DLL ‘‘Grasp’’,0,(‘‘Screw’’,’’yellow’’)
7  DLL ‘‘Grasp’’,1,(‘‘Ledge’’)
8  CALL 1,(0,1)
9  DLL ‘‘Move’’,0,1
10 DLL ‘‘SearchHole’’,0
11 DLL ‘‘PlugIn’’,0
12 DLL ‘‘OpenHand’’,1
13 STOP
```

Da diese Darstellung eines Skriptes sehr unübersichtlich ist, wird ab diesem Punkt der Arbeit anstatt des Skripts das entsprechende Flussdiagramm dargestellt (Abb. 3.8), dessen Knoten die Instruktionen repräsentieren. Die unterschiedliche Form der Knoten

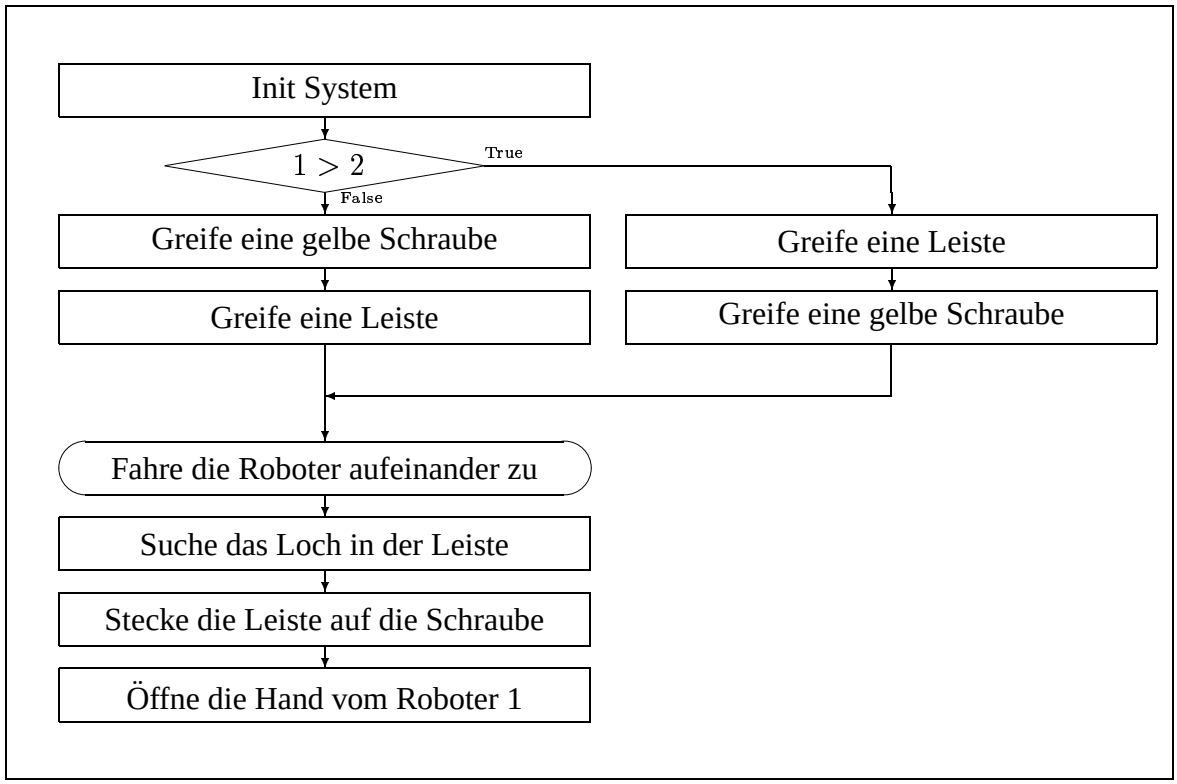


Abbildung 3.8: *Beispielskript als Flussdiagramm.*

verdeutlicht die unterschiedlichen Instruktionen. Ein Rechteck repräsentiert einen Kommandoaufruf, eine Ellipse den Aufruf eines Skriptes und eine Raute eine Verzweigung (IF-THEN Entscheidung). Die Kanten des Graphen verdeutlichen den Kontrollfluss beim Ausführen eines Skriptes durch den Skriptinterpreter.

3.3.3 Ausnahme- und Fehlerbehandlung

Während der Ausführung eines Skriptes kann das Montagesystem zwei Arten von Ausnahmen bzw. Ereignissen verarbeiten:

Synchrone Ereignisse werden vom System selber generiert. Dies ist immer dann der Fall, wenn ein Modul von sich aus bemerkt, dass eine Operation fehlgeschlagen ist. Ob ein *synchrones* Ereignis ausgelöst werden muss, wird an fest definierten Punkten überprüft und das Ereignis bei Bedarf generiert. Z.B. überprüft das Montagesystem durch Testbewegungen, ob eine Steckoperation erfolgreich war; ist dies nicht der Fall, so wird ein *synchrones* Ereignis ausgelöst.

Asynchrone Ereignisse werden von außen generiert, zumeist vom Benutzer, und können zu jedem Zeitpunkt auftreten. Eine solche Intervention vom Benutzer ist z.Z. das *Stopp*-Kommandos, das zu einer sofortigen Unterbrechung der aktuellen Roboteraktion führt.

Beide Arten von Ereignissen werden gleichermaßen behandelt. Ist ein Ereignis aufgetreten, so nimmt das Basissystem dieses Ereignis entgegen und leitet dies an spezielle Module (Kategorie: *Event*) zur Behandlung weiter. Das für das entsprechende Ereignis zuständige Modul entscheidet, wie auf diese Ausnahme zu reagieren ist. Dazu kann auch Rücksprache mit dem Benutzer gehalten werden. Es sind folgende Möglichkeiten vorhanden, auf ein Ereignis zu reagieren.

Abbruch: Die Ausführung des Skriptes wird komplett abgebrochen.

Wiederaufsetzen: Die Ausführung des aktuellen Skriptes wird bei einer zurückliegenden Instruktion fortgesetzt und versucht den fehlgeschlagenen oder abgebrochenen Montageschritt zu wiederholen.

Fehlerbehandlung: Die Ausführung des Skriptes wird bei einer voranliegenden Position fortgesetzt, nachdem zusätzliche Aktionen zur Fehlerbehebung ausgeführt wurden.

Ignorieren: Das Ereignis wird verworfen und die Ausführung des Skriptes bei der folgenden Instruktion fortgesetzt.

Für diese Funktionalität muss die Instruktionssemantik des SI erweitert werden. Die Menge C_0 wird wie folgt erweitert:

$$C_0 \stackrel{\text{def}}{=} \{STOP, Recover@Begin, Recover@End, Catch@Begin, Catch@End\}$$

Zusätzlich werden vier weiter Funktionen benötigt:

$$L_{\text{Recover@Begin}}[i] : \mathbb{Z}^+ \to \mathbb{Z}^+ \tag{3.7}$$

$$L_{\text{Recover@End}}[i] : \mathbb{Z}^+ \to \mathbb{Z}^+ \tag{3.8}$$

$$L_{\text{Catch@Begin}}[i] : \mathbb{Z}^+ \to \mathbb{Z}^+ \tag{3.9}$$

$$L_{\text{Catch@End}}[i] : \mathbb{Z}^+ \to \mathbb{Z}^+ \tag{3.10}$$

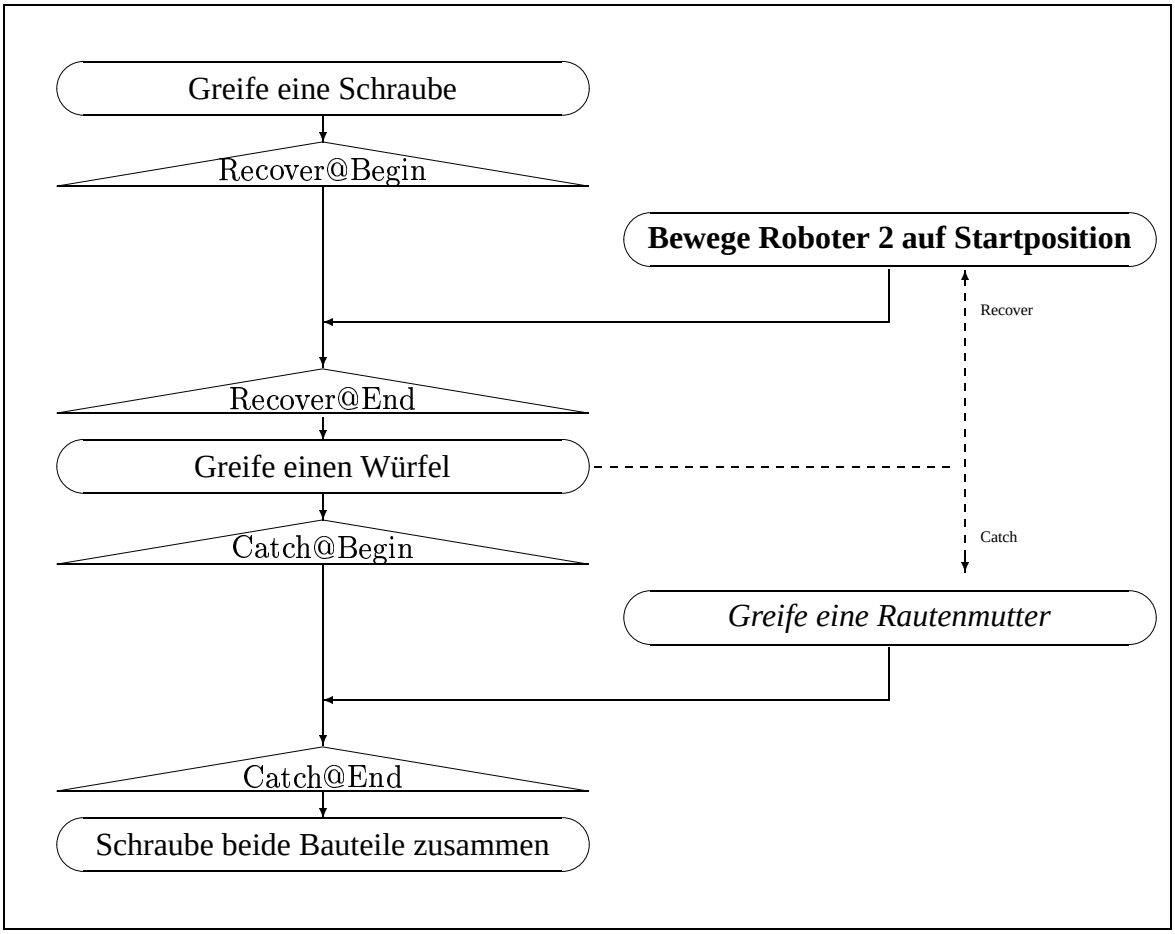


Abbildung 3.9: *Beispielsequenz mit* Recover- *und* Catch-*Umgebung als Fußdiagramm. Die gestrichelten Kanten geben den möglichen Kontrollfluss beim Auftreten eines Ereignisses an.*

Gl. 3.7 liefert die Position des letzten Auftreten einer Operation *Recover@Begin*, ausgehend von der aktuellen Position i. Gl. 3.8 liefert die Position des ersten Auftreten einer Operation *Recover@End*, ausgehend von der aktuellen Position i. Gl. 3.9 liefert die Position des ersten Auftreten einer Operation *Catch@Begin*, ausgehend von der aktuellen Position i. Gl. 3.10 liefert die Position des letzten Auftreten einer Operation *Catch@End*, ausgehend von der aktuellen Position i. Die Semantik der Instruktionen ist:

$$\begin{aligned}
\mathcal{I}[\text{Recover@Begin}](g,z) &\stackrel{\text{def}}{=} (g, L_{\text{Recover@End}}(z)) \\
\mathcal{I}[\text{Recover@End}](g,z) &\stackrel{\text{def}}{=} (g, z+1) \\
\mathcal{I}[\text{Catch@Begin}](g,z) &\stackrel{\text{def}}{=} (g, L_{\text{Catch@End}}(z)) \\
\mathcal{I}[\text{Catch@End}](g,z) &\stackrel{\text{def}}{=} (g, z+1)
\end{aligned}$$

Beschreibt $M_{i,E}$ die Zustandsänderung durch das Modul bis zum Auftreten der Ausnahme, dann ändert sich $\mathcal{I}[DLL\ i,p]$ folgendermaßen:

$$\mathcal{I}[\text{DLL i,p}](g,z) \stackrel{\text{def}}{=} \begin{cases} (M_i(g,p), z+1) & \text{kein Ausnahme} \\ (M_{i,E}(g,p), L_{\text{Recover@Begin}}(z)+1) & \text{Ausnahme mit Wiederaufsetzen} \\ (M_{i,E}(g,p), L_{\text{Catch@Begin}}(z)+1) & \text{Ausnahme mit Fehlerbehandlung} \end{cases}$$

Es können also zwei unterschiedliche Abschnitte zur Behandlung von Ausnahmezuständen in einem Skript definiert werden. Einen *Recover*- und einen *Catch*-Abschnitt. Ein *Recover*-Abschnitt wird **vor** der Instruktion definiert, in der die zu behandelnde Ausnahme auftreten kann. Wird in dieser Instruktion ein Ereignis ausgelöst und wird entschieden, auf dieses Ereignis mit einem Wiederaufsetzen zu reagieren, so wird der letzte *Recover*-Abschnitt **vor** dieser Instruktion ausgeführt. Soll mit einer Fehlerbehandlung auf diese Ausnahme reagiert werden, so wird der erste *Catch*-Abschnitt **nach** der Instruktion ausgeführt[4]. Eine *Catch*- Umgebung für eine Instruktion muss daher **nach** dieser definiert werden. Abb. 3.9 zeigt eine Beispielsequenz mit einer *Recover*- (fett) und einer *Catch*-Umgebung (kursiv).

3.3.4 Sensorzugang

Der Sensorzugang wird ebenfalls über Module realisiert. Die Module, welche Sensorinformationen bereitstellen, gehören der Kategorie SENSOR an. Ein *Aufruf* eines solchen Moduls führt in diesem Fall nicht zur Ausführung einer bestimmten Handlung oder Analyse, sondern zur Aufnahme von Sensordaten. Diese Sensordaten werden dem Aufrufer als Datenpaket geliefert (Abb. 3.10), wobei die Art des Sensors drüber entscheidet, wie das Datenformat beschaffen ist. So wird das Sensormodul für die Kraftsensoren die Daten als sechsdimensionale Vektoren liefern, während ein Modul zur Ansteuerung von Kameras ein entsprechendes Bildformat verwendet. Eine andere Möglichkeit die Sensordaten verfügbar zu machen, besteht in der Hinterlegung der Sensordaten im globalen *Blackboard* (Abb. 3.11). Die Datenübergabe über das *Blackboard* hat den Vorteil, dass die Sensordaten unter einem eindeutigen Namen abgelegt sind und sie dort vom entsprechenden Sensormodul periodisch aktualisieren werden können, ohne dass eine explizite Anforderung durch andere Module erfolgen muss. Zusätzlich hat ein *Sensor*-Modul die Möglichkeit, über spezielle Schnittstellen (siehe Anhang A.2) zur *Benutzerschnittstelle* seine Daten online auf der Benutzeroberfläche zu visualisieren (Abb. A.2). Das Sensormodul kann über eine generische Schnittstelle die Benutzeroberfläche erweitern und darüber den Benutzer die Möglichkeit zu geben die Sensordaten online zu überwachen und sie zu beeinflussen.

[4] Ähnlich einer *try* und *catch*- Umgebung in C++.

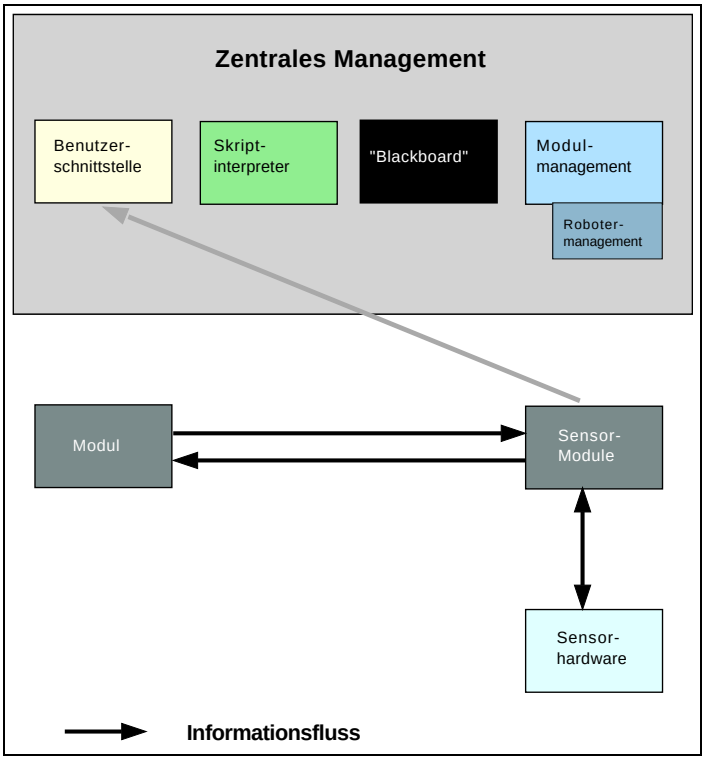


Abbildung 3.10: *Informationsfluss bei direkter Anforderungen von Sensordaten. Ein Modul fordert über einen direkten Funktionsaufruf von einem Sensormodul dessen Daten. Das Sensormodul ermittelt die Sensordaten und liefert sie dem aufrufenden Modul in einer Datenstruktur zurück.*

3.4 Implementationshardware

Die Realisierung von *OPERA* und des darauf aufbauenden Lernverfahrens wurde mit zwei Industrierobotern vom Typ *Puma 260* durchgeführt, welche beide in einer Montagezelle aufgehängt sind. Über einen VME-Bus als Mittler sind die beiden Steuereinheiten der Roboter mit einer SUN Ultra 5 verbunden. Dieser Rechner übernimmt die Trajektorienplanung und gibt den Steuereinheiten in Abständen von 10 ms die einzustellenden Gelenkkonfigurationen vor.

An den *Handgelenken* der Roboter sind sowohl Kraftmomentensensoren (Abb. 3.13 (a) (1)) als auch Handkameras (Abb. 3.13 (a) (2)) montiert. Am Kraftmomentensensor ist ein pneumatischer Zweibackengreifer (Abb. 3.13 (a) (3)) befestigt, dessen Backeninnenseiten mit einem Korkbelag beschichtet ist. Unter den beiden Robotern befindet sich der Montagetisch, von dem die benötigten Montagebauteile von den Manipulatoren gegriffen werden (Abb. 3.13 (a) (4)). Zusätzlich zu den beiden Handkameras der beiden Manipulatoren befinden sich in der Zelle mehrere stationäre Kameras. Für die Objekterkennung und Lageberechnung ist an der Frontseite eine Kamera montiert (Abb. 3.13 (b) (2)), die unter einem Winkel von $45°$ die Szene beobachtet. Zusätzlich existiert ein Ste-

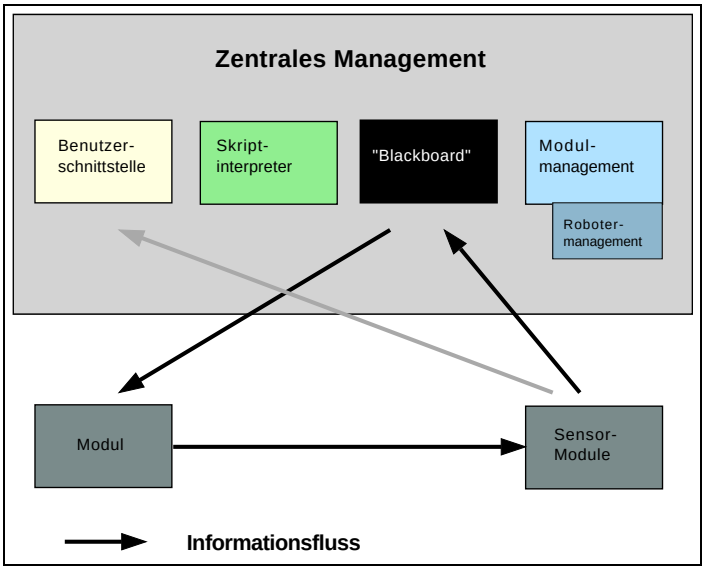


Abbildung 3.11: *Informationsfluss bei Anforderung von Sensordaten über das Blackboard. Statt die Daten dem aufrufenden Modul zu übermitteln werden die Sensordaten im Blackboard abgelegt. Dort kann ein Sensormodul die Daten auch periodisch aktualisieren, ohne explizite Aufforderung.*

reokameraaufbau, der unter demselben Winkel befestigt ist. Eine weitere Kamera (Abb. 3.13 (b) (1)) direkt über den beiden Manipulatoren, ermöglicht einen direkten Blick von oben auf die Szene.

Das Zugrunde liegende Betriebssystem ist SUN Solaris 7, dessen Echtzeitverhalten ausreicht, um die Ausführung des Trajektoriengenerator mit einer Periode von 10 ms zu garantieren. Für die Steuerung der Manipulatoren wurde die C-Bibliothek RCCL [HP86] verwendet, die kartesische- oder Gelenkwinkelinterpolierte Roboterbewegungen zur Verfügung stellt. Zusätzlich ist die Möglichkeit gegeben, die Trajektorie während der Manipulatorbewegung zu ändern, wodurch eine Kraftmomentenregelung ermöglicht wird. *OPERA* selbst ist in der *C++* geschrieben.

3.5 Zusammenfassung

OPERA ist eine Entwicklungs- und Ablaufumgebung, die nur Kernfunktionalitäten für eine Montageanwendung zur Verfügung stellt. Sie besteht aus aus einer graphischen Oberfläche, einem Speicherpool (Blackboard), einem Skriptinterpreter und der Modulverwaltung. Durch Module erhält *OPERA* die eigentliche Montagefunktionalität. Dabei definiert ein Modul in Abhängigkeit von der Kategorie, der sie angehört, eine bestimmte Operation bzw. Funktionalität. Die Anbindung an die Roboter und Sensoren ist ebenfalls durch Modulschnittstellen definiert und die Ansteuerung als Modul realisiert, was eine flexible Anbindung an verschiedene Hardware gestattet. Modulaufrufe können zu

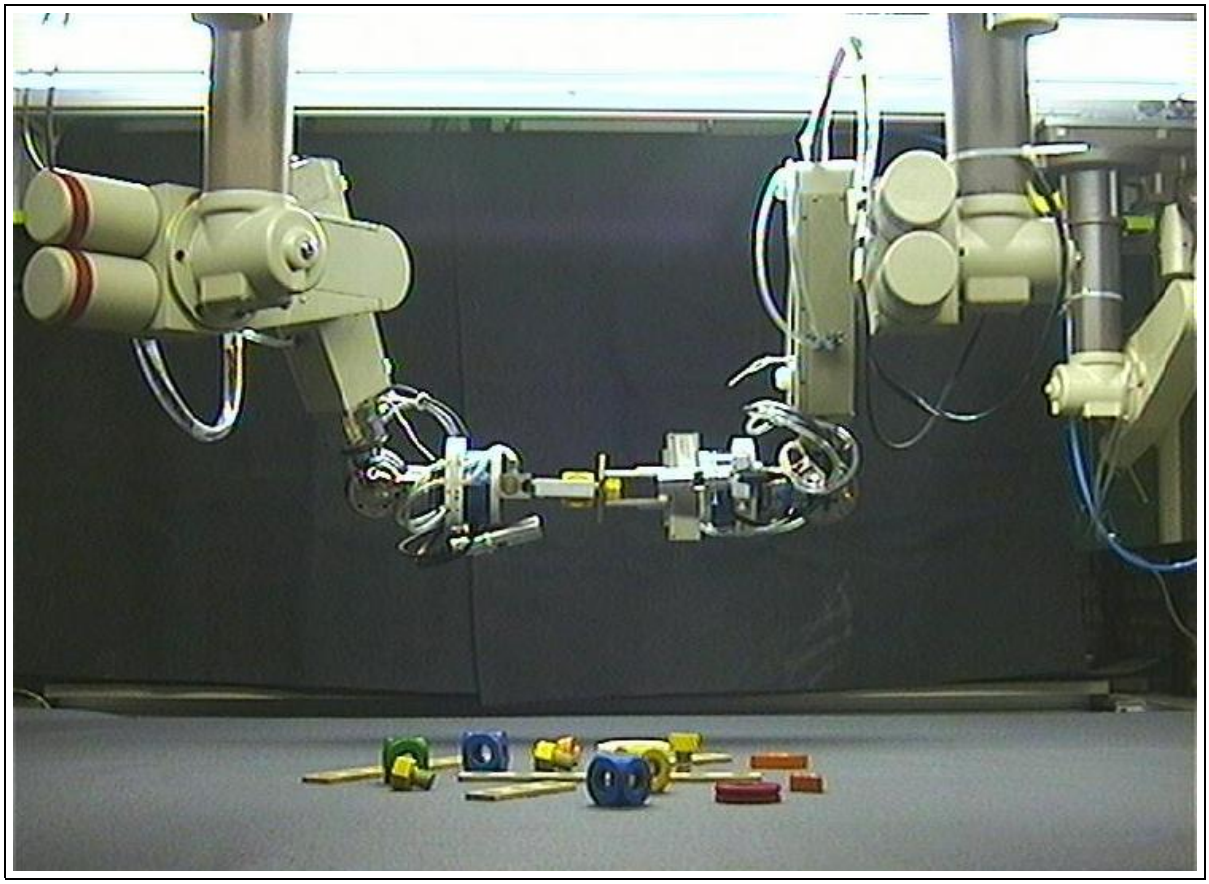
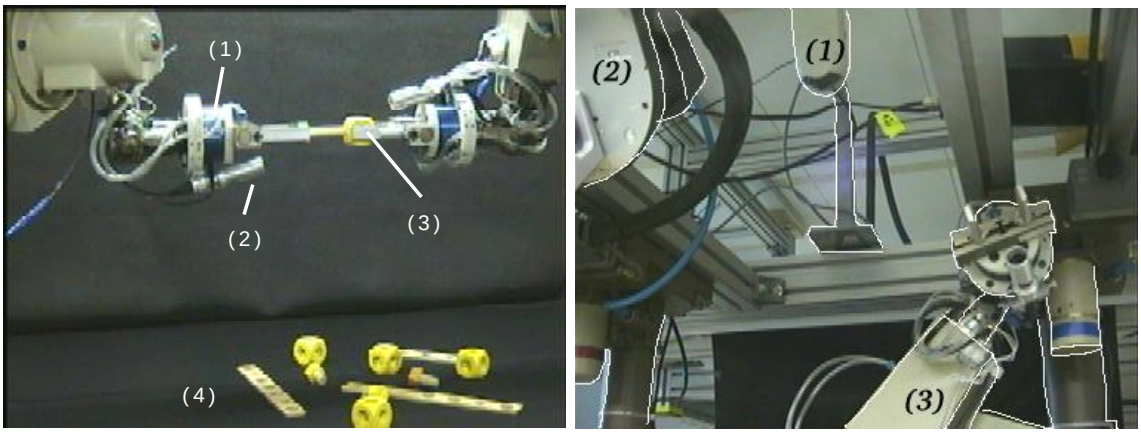


Abbildung 3.12: *Montagezelle mit zwei Manipulatoren.*

Sequenzen bzw. Skripten zusammengestellt werden, die später durch den Skriptinterpreter ausgeführt werden. Die Instruktions- und Programmsemantik wird definiert und erläutert und die Darstellung von Skripten als Flussdiagramm eingeführt. Die Behandlung von Ausnahmezuständen ist im Montagesystem vorgesehen und gestattet deren flexible Bearbeitung.

Die Funktionalität von *OPERA* ist durch den Einsatz im Sonderforschungsbereich 360 demonstriert und Teile dessen Aktorik-Architektur mit *OPERA* realisiert worden. Die Hardwareplattform bilden zwei Puma 260 Manipulatoren, die durch eine SUN Ultra 5 gesteuert werden.

Die Architektur von *OPERA* beinhaltet einen Großteil der in Kapitel 2.1 angesprochenen Merkmale. Es findet eine Komplexitätsreduzierung durch Modularisierung statt wofür Funktionsaufrufe verwendet werden. Eine hierarchische Struktur ist nicht direkt vorgegeben, ist aber möglich. Zusätzlich existiert eine graphische Schnittstelle, über die die entsprechende Programme für den Skriptinterpreter erstellt werden. Mechanismen zur Behandlung von Ausnahmezuständen sind vollständig vorhanden und erlauben sowohl eine Fehlerbehandlung als auch eine Wiederholung der fehlgeschlagenen Operation.



(a) Ausrüstung der Manipulatoren.

(b) Kamerastandorte in der Montagezelle (Roboter in Parkposition (3)).

Abbildung 3.13: *Aufbau der Montagezelle.*

Ein ähnlicher Ansatz zur Realisierung von Roboteroperationen ist in [BKW97] vorgestellt (siehe auch 2.1.3). Die unterste Schicht besteht aus einer Menge von roboterspezifischen Basisoperationen und die drauf aufbauende Schicht beinhaltet einen Interpreter über den die Basisoperationen aktiviert werden. Es fehlt aber ein grafischen Benutzerschnittstelle. Sowohl [JM98] als auch [ZMS97] beschäftigen sich mit der einfacheren Programmierung von Manipulatoren. In [JM98] wird die Modellierung des Arbeitsraumes in ein CAD-Programm verlegt um so die Bahnplanung und Kollisionsvermeidung integrieren zu können. In [ZMS97] wird ein Programm über eine grafische Oberfläche aus vorhandenen Software-Komponenten zusammengestellt. Beide Ansätze generieren zum Abschluss das Programm für die Manipulatoren. Sie überwachen aber die Ausführung des Programms nicht und bilden daher keine integrierte Entwicklungs- und Ablaufumgebung.

Eine solche Umgebung wird in [Smi92] vorgestellt. Die Programmierung der Anwendung erfolgt ausschließlich über die Programmiersprache C, die entweder übersetzt oder interpretiert wird. Sowohl einfache wie auch komplexe Befehle zur Ansteuerung der Roboter werden zur Verfügung gestellt. Daten werden in einer Datenbank gespeichert. Bekannte Datenstrukturen können über eine Benutzerschnittstelle zur Datenbank komfortabel kontrolliert und verändert werden, so dass auch auf die Sensordatenauswertung einfach Einfluss genommen werden kann. Es existiert allerdings kein allgemeines Konzept, wie neue Sensoren ins System integriert werden. Die Robotersteuerung kann ebenfalls über die grafische Benutzeroberfläche vorgenommen werden. Da aber das System Trajektorienberechnung und Kollisionsvermeidung durchführt, wurde sich auf einen Roboter mir vier Freiheitsgraden beschränkt. Eine Konzept zur Steuerung von anderen Robotern ist nicht vorhanden.

Kapitel 4

Kompakte Darstellung von Sensormustern

4.1 B-Spline Fuzzy Regler

Beim Aufbau eines flexiblen Montagesystems hat sich die Verwendung eines B-Spline Fuzzy Reglers [ZK96, ZL96] in verschiedenen Anwendungen bewährt. In [ZFK00, ZF98, ZvCK97] wird dieser Ansatz als Regler zur Steuerung der in Kapitel 3.4 beschriebenen Manipulatoren verwendet. Schraub-, Steck- und Mehrarmtrageoperationen sind damit verwirklicht worden. Bei diesen Operationen ist die Fähigkeit des B-Spline Fuzzy Reglers, sich schnell und insbesondere während des Betriebes zu adaptieren, ein großer Vorteil. Die Verwendung von Reglerparametern einer Trageoperation in einem Regler für einen Schraubvorgang [FZK99] zeigt die Flexibilität des B-Spline Fuzzy Ansatzes bei gleichen physikalischen Gesetzmäßigkeiten. Es konnte gezeigt werden, dass mit Hilfe des B-Spline Fuzzy Reglers als mehrdimensionalem Funktionsapproximator sowohl *Visual Servoing* [ZKS00, ZKS99, ZSK99] , z.B. das Ausrichten eines Greifer über einem Bauteil, als auch Datenfusion [vCSZK99, vCFZK00] betrieben werden kann. Diese Eigenschaft steht auch in diesem Kapitel im Vordergrund.

4.1.1 B-Spline Basisfunktionen

I.J. Schoenberg [Sch46] hat B-Splines zur Glättung von statistischen Daten und R.F. Riesenfeld [Rie73] und W.J. Gordon [GR74] haben sie zur Repräsentation von Kurven und Oberflächen verwendet. B-Splines basieren auf Polynomen niedriger Ordnung und lassen sich einfach berechnen. Obwohl B-Splines meistens in der offline Modellierung verwendet werden und die Fuzzy Technik sich mit online Steuerung befaßt, lassen sie sich doch auch in Fuzzy-Reglern verwenden. Zhang und Raczkowsky haben in [ZRH94] gezeigt, daß sowohl B-Spline Basisfunktionen als auch Zugehörigkeitsfunktionen von linguistischen Variablen normalisierte und überlappende Funktionshüllen sind. Beide Typen von Funktionen besitzen gute Eigenschaften zur Interpolation [ZK96]. Der Grundgedanke besteht darin, die Zugehörigkeitsfunktionen der linguistischen Terme mit B-Spline Basisfunktionen zu modellieren.

Sei

$$\xi = (x_0, \ldots, x_k)$$

ein sog. *Knotenvektor*.

Definition 4.1.1 *Eine B-Spline Basisfunktion ist definiert als*

$$N_i^1(x) = \begin{cases} 1 & \text{für } x \in [x_i, x_{i+1}) \\ 0 & \text{sonst} \end{cases}$$

und für $n > 1$

$$N_i^n(x) = \frac{x - x_i}{x_{i+n-1} - x_i} \cdot N_i^{n-1}(x) + \frac{x_{i+n} - x}{x_{i+n} - x_{i+1}} \cdot N_{i+1}^{n-1}(x) \ .$$

n *bezeichnet die Ordnung der Basisfunktion und gibt die Anzahl der Intervalle an, in denen* $N_i^n(x) > 0$ *ist.*

Die Abb. 4.1 zeigt B-Spline Basisfunktionen verschiedener Ordnung, die auf dem Knotenvektor $(0, 1, 2, 3, 4, 5)$ basieren. Abb. 4.2(a) und 4.2(b) zeigen alle auf dem Knoten-

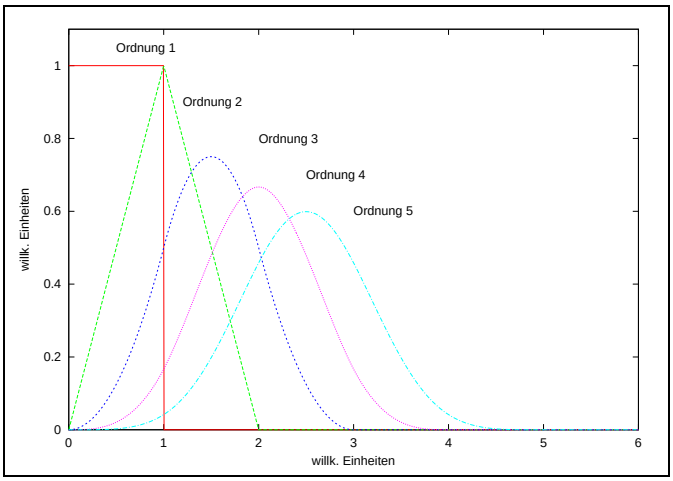


Abbildung 4.1: *B-Spline Basisfunktionen mit unterschiedlicher Ordnung.*

vektor $\xi = (0, 1, 2, 3, 4, 5)$ möglichen Basisfunktionen mit der Ordnung 2 und 3.

Wichtige Eigenschaften

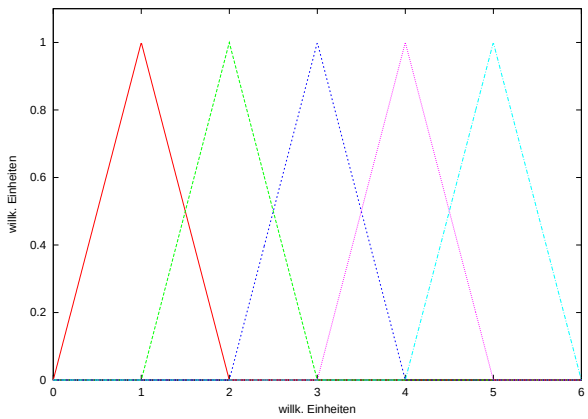
Partition der 1 (*Partition of Unity*):

$$\sum_{i=0}^{k-n} N_i^n(x) = 1, \quad \forall x \in [x_{n-1}, x_{k-n+1}] \tag{4.1}$$

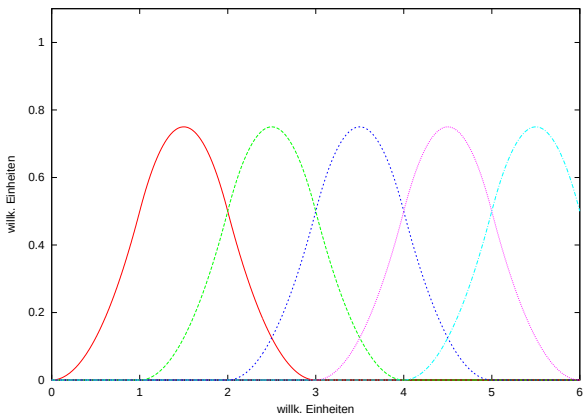
Differenzierbarkeit: Gilt

$$x_i \neq x_j \quad \forall i, j \in \{0, \ldots, k\}$$

dann ist $N_i^n \in C^{n-2}$, d.h. N_i^n ist $n-2$-mal stetig differenzierbar.



(a) B-Spline Basisfunktionen $N_0^2, N_1^2, N_2^2, N_3^2, N_4^2$.



(b) B-Spline Basisfunktionen $N_0^3, N_1^3, N_2^3, N_3^3$.

Abbildung 4.2: *B-Splines der Ordnung zwei und drei.*

B-Spline Basisfunktionen als Zugehörigkeitsfunktionen

In der Praxis besitzt die Glattheit der Reglerausgabe in Bezug auf den Verschleiß der zu regelnden Maschinen eine wichtige Bedeutung, da von ihr oft die Lebenserwartung der Bauteile abhängt. Bei der oft verwendeten Dreiecksfunktion als Zugehörigkeitsfunktion ist zwar die Ausgabe des Reglers stetig, aber nicht mehr stetig differenzierbar.

Bei Verwendung der B-Spline Basisfunktionen als Zugehörigkeitsfunktionen muss der Eingabewertebereich $[a, b]$ in die gewünschte Anzahl von Intervallen eingeteilt werden. Die Anzahl der Knotenpunkte ergibt sich dann aus der Anzahl der linguistischen Terme m und der Ordnung der verwendeten Basisfunktionen. Da im Folgenden von der Gültigkeit der *Partition of Unity* (Gleichung (4.1)) ausgegangen wird, besitzt der Knotenvektor folgende Form:

$$\xi = (x_0, \ldots, x_{n-1}, \ldots, x_{n+m-1}, \ldots, x_{m+2n-2})$$

mit

$$x_{n-1} = a$$
$$x_{m+2n-2} = b$$

Beispiel

Das Intervall $[-2, +2]$ soll mit vier linguistischen Termen von B-Spline Basisfunktionen der Ordnung 3 beschrieben werden. Der Knotenvektor sieht bei äquidistanten Knotenpunkten folgendermaßen aus:

$$\xi = (-4, -3, -2, -1, 0, 1, 2, 3, 4)$$

Abb. 4.3 zeigt die B-Spline Basisfunktionen als Zugehörigkeitsfunktionen (ZF'n) sowie die Gültigkeit von Gleichung (4.1) in Intervall $[-2, 2]$ gegeben ist (*Summe aller ZF'n*). Zusätzlich zu den gewünschten Basisfunktionen bzw. ZF'n werden am rechten und lin-

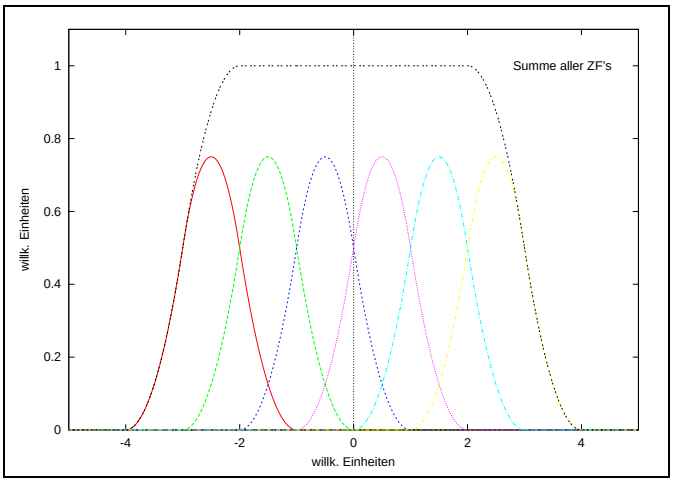


Abbildung 4.3: *B-Spline Basisfunktionen zum Knotenvektor ξ und ihre Summe* $\sum_{i=0}^{k-n} N_i^n(x)$.

ken Rand zwei weitere Basisfunktionen benötigt. Diese haben den Zweck die *Partition of Unity* über das gesamte Intervall $[-2, +2]$ zu gewährleisten. Sie beschreiben die sogenannten virtuellen linguistischen Terme. Verallgemeinert bedeutet dies:
Soll das gewünschte Eingabeintervall $[a, b]$ mit m linguistischen Termen abgedeckt werden und die *Partition of Unity* gegeben sein, dann werden benötigt:

- $m + n - 2$ Knotenpunkte für das Intervall $[a, b]$, wobei a und b die äußeren Knotenpunkte sind.
- an jeder Seite des Intervalls $n - 1$ zusätzliche Knotenpunkte.
- m B-Spline Basisfunktionen
- und zusätzlich an jeder Seite des Intervalls $n - 2$ B-Spline Basisfunktionen.

$A_1 \dots A_m$ bezeichnen die linguistischen Terme und $A_{m+1}, \dots, A_{m+n-2}$ bzw. $A_{-n+3}, \dots, A_0$ die virtuellen linguistischen Terme.

Konstruktion eines Single Input Single Output (SISO) Systems

Sei x die Eingangsvariable des Systems, y die Ausgangsvariable und

$$\text{Regel}(i): \text{ IF } x \text{ is } A_i \text{ THEN } y \text{ is } y_i \ .$$

Dann ist die Menge der Kernregeln *CRS* gegeben durch

$$CRS = \{\text{Regel}(i) \quad | \quad i = 1, \dots, m\} \tag{4.2}$$

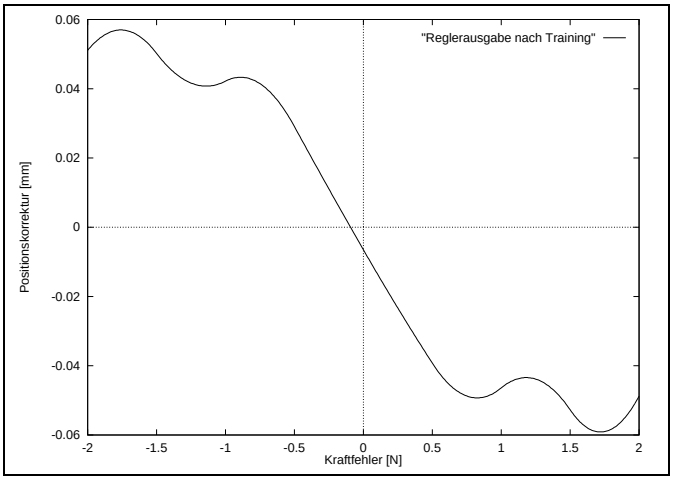


Abbildung 4.4: *Mögliche Ausgabekurve eines Fuzzy-Reglers mit B-Funktionen als ZF'n zur Regelung der Andruckkraft während eines Schraubvorganges.*

Alle Regeln, die über die Rand-B-Funktionen gemacht werden, heißen Randregeln und gehören zu der Menge der Randregeln *MRS*. Sei

$$\text{Regel}_1(i) : \text{IF } x \text{ is } A_i \text{ THEN } y \text{ is } y_1 \tag{4.3}$$

$$\text{Regel}_m(i) : \text{IF } x \text{ is } A_i \text{ THEN } y \text{ is } y_m \tag{4.4}$$

dann ist

$$MRS = \{\text{Regel}_1(i) | \quad i = -n+3, \ldots, 0\} \quad \bigcup \tag{4.5}$$

$$\{\text{Regel}_m(i) | \quad i = m+1, \ldots, m+n-2\} \tag{4.6}$$

Die Eingabe x wird als Singleton interpretiert und die Ausgänge der Randregeln werden auf die erste bzw. letzte Kernregel abgebildet.
Die Vereinigung von *CRS* und *MRS* ergibt den gesamten Regelsatz

$$RS \; = \; CRS \bigcup MRS$$

In der Fuzzy-Logik gelten immer alle Regeln gemäß ihres Zutreffens und es werden bei diesen Regler alle *THEN*-Teile addiert. Mit der Defuzzifizierung *Zentrum des Durchschnitts* (centre average) ist die Ausgabe:

$$y = \frac{\sum_{i=0}^{m+n-2} y_i N_i^n(x)}{\sum_{i=0}^{m+n-2} N_i^n(x)}$$

Da aber x nur aus dem Intervall $[a, b]$ zugelassen ist, und dort die *partition of unity* gilt, folgt:

$$y = \sum_{i=0}^{m+n-2} y_i N_i^n(x)$$

Abb. 4.4 zeigt eine mögliche Ausgabekurve eines solchen B-Spline Fuzzyreglers.

Konstruktion eines Multiple Input Single Output (MISO) Systems

Werden (4.2) und (4.6) verallgemeinert, so ergibt sich:

$$\text{Regel}(i):\ \text{IF}\ (x_1\ \text{is}\ A_{i_1})\ \text{and}\ \ldots\ \text{and}\ (x_q\ \text{is}\ A_{i_q})\ \text{THEN}\ y\ \text{is}\ y_{i_1,\ldots,i_q}$$

und für die Ausgabe gilt:

$$y = \sum_{i_1=0}^{n_1+m_1-2} \cdots \sum_{i_q=0}^{n_q+m_q-2} y_{i_1,\ldots,i_q} \prod_{j=1}^{q} N_{i_j}^{n_j}(x_j)$$

Iteratives Ermitteln der Singletons

Üblicherweise sind die konkreten Werte der Singletons a priori nicht gegeben. Ist die zu interpolierende Funktion bekannt, können die Singletons durch Aufstellen eines entsprechenden Gleichungssystems ermittelt werden. Ist die zu interpolierende Funktion nicht bekannt, wie z.B. in [ZvCK97], so können die Werte iterativ über Beispiele *gelernt* werden.

Das Verfahren basiert auf einem Gradientenabstiegsverfahren unter Verwendung der quadratischen Fehlerfunktion.

$$E = \frac{1}{2} \sum (\text{Erwarteter Wert} - \text{Reglerausgabe})^2$$

In jedem Lernschritt werden die Singletons wie folgt geändert:

$$\delta_{y_i} = (\text{Erwarteter Wert} - \text{Reglerausgabe})\, N_i^n$$

Jeder Singleton wird um δ_{y_i} erhöht. Die Änderung besteht aus der Differenz zwischen der Reglerausgabe und der erwarteten Ausgabe, welche mit dem Funktionswert der korrespondierenden B-Spline Basisfunktion gewichtet wird. Selbst wenn die Differenz nur qualitativ bekannt ist, kann die Lernregel unter Verwendung eines weiteren Faktors, der die Lernschrittweite angibt, angewendet werden [ZvCK97].

Da die B-Spline Basisfunktionen nur auf n Intervallen einen Funktionswert ungleich Null besitzen, *feuern* nur n Regeln gleichzeitig. Dies begrenzt nicht nur den Berechnungsaufwand, da maximal n Funktionswerte pro Dimension berechnet werden müssen, sondern hat zur Folge, dass die Änderung eines Singletons nur eine lokale Änderung der Ausgabefunktion bewirkt. Der Regler bleibt nach dieser Änderung insgesamt verwendbar, da andere Wertebereiche von dieser Änderung nicht betroffen sind. Die Verwendung der quadratischen Fehlerfunktion bewirkt, dass das Trainingsverfahren im globalen Minimum konvergiert (siehe [ZK96]).

4.1.2 Sensordatenfusion mit einem B-Spline Fuzzy Regler

Darüber hinaus kann der B-Spline Fuzzy Regler zur Sensordatenfusion durch Zusammenfassen aller Größen zur Reglereingabe verwendet werden. Der B-Spline Fuzzy Regler erhält dann nicht mehr einen Sensortyp als Eingabe, sondern unterschiedliche und berechnet daraus die Ausgangsgröße. Ein Beispiel ist in Abb. 4.5 gegeben und zeigt die in [ZF98] verwendeten Reglereingabegrößen. Dort werden Kraftmomentendaten und Positionsdaten als Eingabewerte für den B-Spline Fuzzy Regler verwendet, ohne sie vorzuverarbeiten. Die Ausgabe ist die Positionskorrektur eines Manipulators für eine Dimension. Die Voraussetzung für die Anwendbarkeit dieses Fusionskonzeptes ist die Annahme, dass ähnliche Situationen in der realen Welt im Sensorraum[1] einen geringeren Abstand haben als unähnliche Situationen. Dies trifft für reale kontinuierliche Systeme zu.

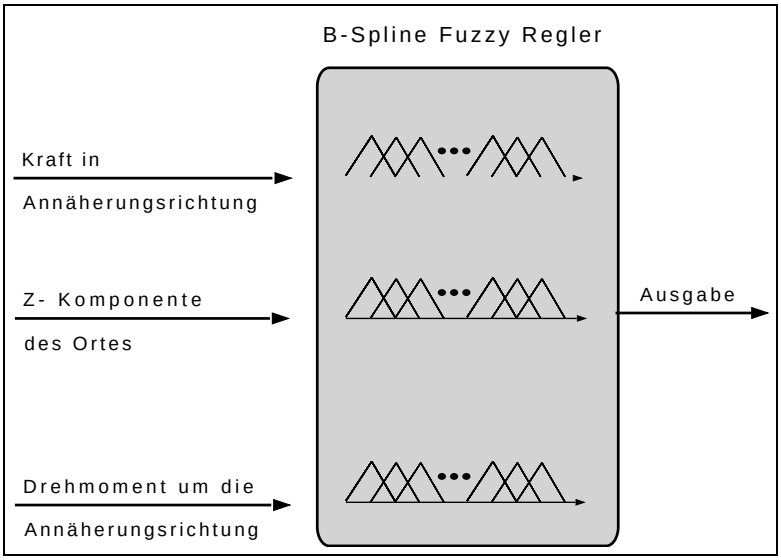


Abbildung 4.5: *Exemplarische Sensordatenfusion.*

Die Verwendung des B-Spline Fuzzy Regler als Approximator bietet sich an, da

- die Differenzierbarkeit der Reglerausgabekurve durch die Ordnung der B-Spline Basisfunktionen festgelegt wird. Die Reglerausgabekurve ist $n-2$ mal stetig differenzierbar.
- die Generierung der ling. Terme durch die Verwendung der rekursiven Definition erfolgt, so dass nur die Anzahl der Terme für jede Dimension vorgegeben werden muss.
- die Anzahl der Singletons identisch mit der Anzahl von Regeln ist, so dass Vorwissen einfach eingebracht werden kann.

[1] Raum, den die Sensordaten aufspannen.

- das Trainingsverfahren im globalen Minimum konvergiert.

4.2 Hauptkomponentenanalyse

Nimmt die Zahl der zu fusionierenden Sensordaten zu, so ist der oben beschriebene Ansatz theoretisch weiter gültig, aber in der realen Anwendung nicht zu mehr zu realisieren. Die Zahl der Singletons des zu berechnenden Reglers mit n Eingangsgrößen und m B-Splines pro Eingangsgröße beträgt m^n. So besitzt beispielsweise ein Regler mit zehn Eingangsgrößen, die mit jeweils zehn B-Splines abgedeckt werden, 10^{10} Singletons. Wird der Wert eines Singletons mit einfacher Genauigkeit gespeichert, so besteht ein Speicherbedarf von 37 GByte.

Werden Bilder als Eingangsdaten verwendet, so werden zehn Eingangsgrößen sofort überschritten. Andererseits enthalten Bilder aber viele Informationen, die nicht oder nur sehr wenig zur Beurteilung der aktuellen Situation beitragen. Es liegt daher nahe, die Eingangsdaten auf die Größen zu reduzieren, die auch wirklich die relevanten Informationen beinhalten. Es gibt zwei Hauptverfahren, um das Problem der Dimensionsreduzierung zu lösen: *Input selection* [JSM97, Chi96] und Hierarchie. *Input selection* ist eine experimentelle Methode und besteht darin, durch heuristische Suche die wichtigsten Eingangsgrößen zu ermitteln. Dieses Verfahren ist nicht nur sehr aufwendig, sondern beinhaltet auch einen gewissen Informationsverlust, da nur die wichtigsten Größen beachtet werden.

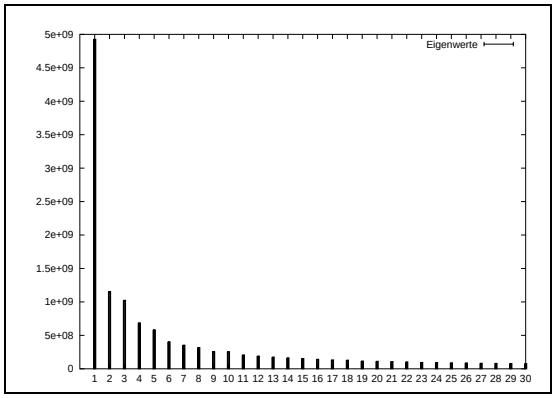


Abbildung 4.6: *Die sortierten Eigenwerte einer durch Bilder gebildeten Kovarianzmatrix.*

Die hierarchische Strukturierung [LT97] der Daten setzt voraus, dass sie sich auch in Gruppen unterteilen lassen. Allerdings existiert kein allgemeines Verfahren, um diese Strukturierung vorzunehmen.

Ein bekanntes Mittel, um in der Statistik mit hochdimensionalen Problemen umzugehen, ist die *Principal Component Analysis (PCA)* [2]. Wie in [NMN94] gezeigt wird, ist

[2] Die PCA ist ein Verfahren, um aus einer Menge von abhängigen Zufallsvariablen unabhängige Linearkombinationen zu bestimmen, wobei die Linearkombinationen mit großer Varianz *principal components* heißen.

diese Dimensionsreduzierung in einen Unterraum auch bei allgemeinen Regelproblemen anwendbar. Es wird ein Unterraum der Dimension M aufgespannt, dessen Achsen in Richtung der größten Varianz der Eingangsdaten zeigen. Dieses sind die M Eigenvektoren zu den korrespondierenden M größten Eigenwerten. Dieser Raum wird im weiteren als Eigenraum bezeichnet. In diesen Eigenraum werden die Eingangsdaten abgebildet und Abbildung wird als Eingangsdatum für den B-Spline Fuzzy Regler verwendet. Die Anzahl der benötigten Eigenvektoren ist abhängig vom jeweiligen Problem und muss so groß sein, dass die notwendige Information bei der Projektion in den Eigenraum erhalten bleibt. Hat der Eingaberaum die Dimension N, so ist üblicherweise $M \ll N$. Als Beispiel zeigt Abb. 4.6 die ersten 30 nach Größe sortierten Eigenwerte aus der in [vCZK98] vorgestellten Problemstellung. In dieser Arbeit wird eine Schraubenspitze über das Loch einer Leiste mittels Kameradaten (Abb. 4.7) geführt. Es ist gut

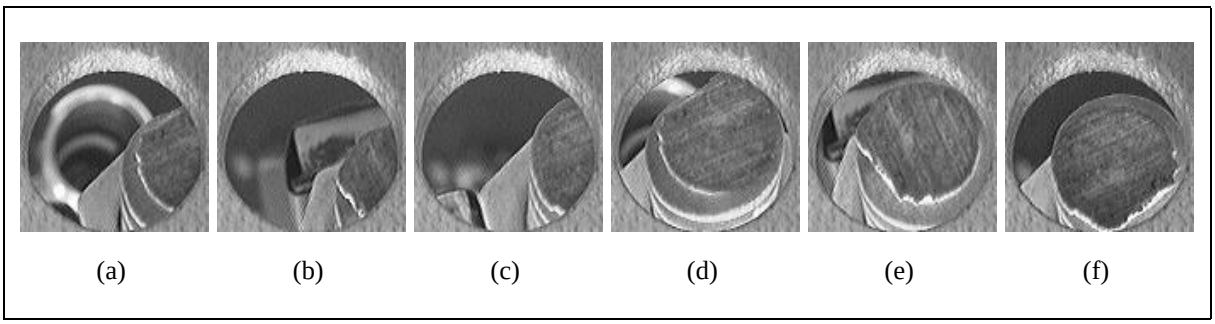


Abbildung 4.7: Sichten einer Handkamera während der Steuerungsoperation (Bildgröße: 111×103). Die Bilder zeigen das Loch in der Leiste. Durch das Loch ist die Schraubenspitze zu sehen, die über dem Loch zentriert werden soll. Teilweise sind ebenfalls Teile des Manipulators zu sehen, der die Schraube hält (z.B. die Handkamera in Bild (a)).

zu sehen, dass nur für die ersten Eigenvektoren eine nennenswerte Varianz in den Eingangsdaten besteht. Das Konzept besteht nun darin, hochdimensionale Eingangsdaten in den Eigenraum abzubilden und auf die Funktionswerte einen B-Spline Fuzzy Regler zu verwenden (siehe Abb. 4.8). Der Regler kann auch als eine Schicht eines neuronalen Netzwerkes angesehen werden. Die größte Varianz der Daten gehört zum Eigenvektor mit dem größten Eigenwert und deshalb wird die Abbildung auf dieser Achse mit einer größeren Anzahl von B-Splines abgedeckt. Je kleiner die Varianz, um so weniger B-Splines bzw. ling. Terme sind notwendig, um diese Dimension des Eigenraums abzudecken.

Adaptive Berechnung der PCA

Es gibt verschiedene Methoden aus einer Datenmenge die Eigenvektoren zu bestimmen. In [Sch98b] ist ein Verfahren zur adaptiven Berechnung der Eigenwerte vorgestellt worden. Es basiert auf einem Perzeptronnetzwerk und wird über die *Hebb'sche Lernregel* trainiert. Die dort vorgestellte Variante der Lernregel entspricht der von Yuille et

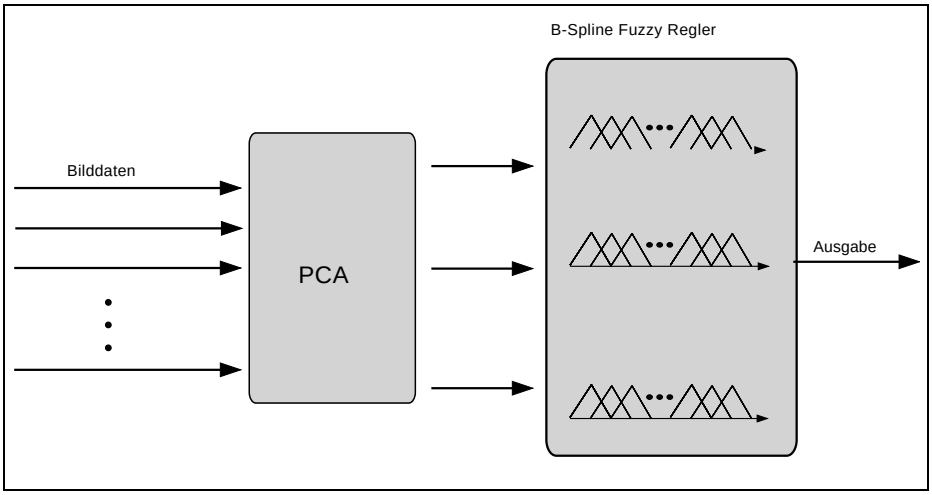


Abbildung 4.8: *Die Struktur eines Fuzzy Reglers mit vorgeschalteter Reduktion über PCA.*

al [YKC89]:

$$V = \sum w_j \xi_j$$

Der Netzausgang V wird durch die Projektion des Eingangsvektors $\vec{\xi}$ auf den Gewichtsvektor $\vec{w}$ gebildet. Die Veränderung des Gewichtsvektors wird wie folgt vorgenommen:

$$\Delta w_j = \eta(V\xi_j - w_j|\vec{w}|^2) \tag{4.7}$$

η bezeichnet die Lernschrittweite. Der Gewichtsvektor $\vec{w}$ zeigt im Konvergenzfall in die Richtung des Eigenvektors mit dem größten Eigenwert λ_{max}. Dabei geht $|\vec{w}|$ monoton steigend gegen $\sqrt{\lambda_{\text{max}}}$.

Soll mehr als ein Eigenvektor berechnet werden, so wird nach dem Berechnen des ersten Eigenvektors von jedem Eingangsdatum dessen Projektion auf den Eigenvektor subtrahiert. Es wird dadurch ein Subraum gebildet, der orthogonal zum ersten Eigenvektor liegt und von dem sich der nächste Eigenvektor berechnen lässt. Dieses Verfahren lässt sich so lange anwenden, bis die gewünschte Anzahl von Eigenvektoren berechnet worden ist.

Die Eingangsdaten selbst erhält man durch die Bildung einer Matrix bestehend aus den einzelnen Trainingsbeispielen und Berechnen der Kovarianzmatrix.

Diskrete Berechnung der PCA

In [Sch98a] wird die Berechnung der Eigenwerte auf Grundlage des *Jacobi-Verfahrens* durchgeführt. Es wird aber nicht die Kovarianzmatrix gebildet, sondern die implizite Kovarianzmatrix. Dies hat den Vorteil, dass bei n Trainingsbeispielen der Dimension m eine $n \times n$ Matrix verarbeitet werden muss und nicht eine $m \times m$ Matrix, da üblicherweise $m \gg n$. Aus den über die impliziten Kovarianzmatrix berechneten Eigenwerte $\tilde{\lambda}$

und -vektoren $\tilde{\vec{a}}$ können die Eigenwerte und Eigenvektoren folgendermaßen berechnet werden:

$$\begin{aligned} \lambda_i &= \tilde{\lambda}_i \\ \vec{a}_i &= \tilde{\lambda}_i^{-\frac{1}{2}} \; X \; \tilde{\vec{a}}_i \end{aligned}$$

Wobei X die aus den Trainingsbeispielen gebildete Matrix

$$X = \left[(\vec{x}^1 - \vec{\mu}) \ldots (\vec{x}^m - \vec{\mu}) \right]$$

und $\vec{\mu}$ der Mittelwertvektor ist. Die eigentliche Berechnung der Eigenwerte erfolgt über das LR-Verfahren.

Algorithmus 4.2.1 *LR-Verfahren*

Gegeben: *(n,n)-Matrix A.*

Gesucht: *Sämtliche Eigenwerte $\lambda_i, i = 1, \ldots, n$, von A*

1. *Setze $A_1 := A$.*
2. *Führe für jedes $i = 1, 2, 3, \ldots$ durch:*
   (a) *Die Faktorisierung $A_i = L_i R_i$ (sofern durchführbar) mit einer normierten unteren Dreiecksmatrix L_i und einer oberen Dreiecksmatrix R_i.*
   (b) *Die Matrixmultiplikation $A_{i+1} = R_i L_i$. Die Matrizen A_i sind ähnlich zu A.*

Dann gilt unter gewissen Voraussetzungen

$$\lim_{i \to \infty} A_i = \lim_{i \to \infty} R_i = \begin{pmatrix} \lambda_i & \cdots & * \\ & \ddots & \vdots \\ 0 & & \lambda_n \end{pmatrix} \text{ und } \lim_{i \to \infty} L_i = E$$

Da das LR-Verfahren nicht immer durchführbar ist, wurden die Eigenwerte für die vorliegende Arbeit mit dem in [EMR96] vorgestellten Verfahren berechnet. Es beruht auf dem QR-Verfahren. Die QR-Zerlegung einer Matrix A mit Hilfe der Householder-Transformation ist die Grundlage für das QR-Verfahren.

Satz 4.2.1 *Sei $\vec{v} \in \mathbb{G}^n$ ein Vektor und E die (n,n)-Einheitsmatrix. Dann ist*

$$H := E - \frac{2}{||v||^2} v v^T$$

eine symmetrische, orthogonale (n,n)-Matrix (Householder-Matrix), d.h. es gilt $H^T H = H^2 = E$.

Die QR-Zerlegung ist dann eindeutig, wenn die (n,n)-Matrix A nichtsingulär ist und die Vorzeichen der Diagonalelemente der Superdiagonalmatrix R fest vorgeschrieben sind.

Algorithmus 4.2.2 *QR-Verfahren von Rutishauser*

Gegeben: *(n,n)-Matrix A.*

Gesucht: *Sämtliche Eigenwerte von A*

1. *Setze $A_1 := A$*
2. *Führe für jedes $i = 1, 2, 3, \ldots$ folgende Schritte durch:*
   (a) *Faktorisierung $A_i = Q_i R_i$ mit der unitären Matrix Q_i (d.h. $Q_i^{-1} = \bar{Q}_i^T$) und der oberen Dreiecksmatrix R_i.*
   (b) *Die Matrizenmultiplikation $A_{i+1} = R_i Q_i$*

Dann gilt unter gewissen Voraussetzungen (etwa für $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n| > 0$)

$$\lim_{i\to\infty} A_i \begin{pmatrix} \lambda_1 & \cdots & * \\ & \ddots & \\ 0 & & \lambda_n \end{pmatrix}$$

Es ist empfehlenswert, die (n,n)-Matrix auf obere Hessenbergform zu transformieren, um den erheblichen Rechenaufwand herabsetzen zu können. Jede (n,n)-Matrix $A = (a_{ik})$, $a_{ik} \in \mathbb{G}$, lässt sich mit Hilfe von symmetrischen, orthogonalen Householdermatrizen auf obere Hessenbergform $\tilde{A}$ transformieren:

$$\tilde{A} = \begin{pmatrix} * & * & * & \cdots & * \\ * & * & * & \cdots & * \\ 0 & * & * & \cdots & * \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & * & * \end{pmatrix} = (\tilde{a}_{ik})$$

und $\tilde{a}_{ik} = 0$ für $i \geq k + 2$. Ist A symmetrisch, d.h. $A^T = A$, so ist die zugehörige Hessenbergmatrix symmetrisch und tridiagonal.

Algorithmus 4.2.3 *Transformation auf Hessenbergform*

Gegeben: $A_1 = (a_{ik}^{(1)}) := A, i, k = 1, \ldots, n$

Gesucht: *A auf obere Hessenbergform transformieren.*

Für jedes $i = 1, \ldots, n-2$ sind dann folgende Schritte auszuführen:

1. *Berechnung der (n-i)-reihigen Householdermatrix H_i nach der Vorschrift*

$$\tilde{H}_i = E_{n-i} - \frac{2}{||v_i||^2} v_i v_i^T$$

mit

$$v_i = \begin{pmatrix} a_{i+1,i}^{(i)} & +sign(a_{i+1,i}^{(i)}))||a_i^{(i)}|| \\ a_{i+2,i}^{(i)} & \\ \vdots & \\ a_{n,i}^{(i)} & \end{pmatrix}, a_i^{(i)} = \begin{pmatrix} a_{i+1,i}^{(i)} \\ a_{i+2,i}^{(i)} \\ \vdots \\ a_{n,i}^{(i)} \end{pmatrix}$$

2. *Man setze*

$$H_i = \begin{pmatrix} E_i & 0 \\ 0 & H_i \end{pmatrix} \begin{matrix} \} & i \\ \} & n-1 \end{matrix}$$

und berechne $A_{i+1} := H_i A_i H_i = (a_{jk}^{(i+1)})$.

Dann besitzt A_{n-1} obere Hessenbergform und der Rechenaufwand beträgt $(5/3)n^3 + O(n^2)$ Punktoperationen.

Trainingsverfahren

Das Verfahren zur Realisierung eines Reglers besteht dann aus zwei Teilschritten: Der Bestimmung der *principal components* aus den aufgezeichneten Trainingsbeispielen und des Trainings eines B-Spline Fuzzy Reglers als Funktionsapproximator. Handelt es sich bei den Trainingsbeispielen um Bilder, so sollten diese in der Energie normiert werden, damit Einflüsse aufgrund von Helligkeitsschwankungen vermieden werden, wobei das Bild als Vektor $\vec{b}$ betrachtet wird.

$$\sum_{i=0}^{n} b_i = 1$$

Nach dem Abzug des Durchschnittsbildes, wodurch nur noch die Unterschiede in den einzelnen Trainingsmustern verbleiben, wird die Kovarianzmatrix bzw. implizite Kovarianzmatrix gebildet und die Eigenwerte und Vektoren iterativ oder direkt berechnet. Nach Wahl der Anzahl der benötigten Eigenvektoren werden die Trainingsmuster in den Eigenraum projiziert. Der neu entstandene Satz von Trainingsmustern bildet die Trainingsdaten für den B-Spline Fuzzy Regler.

Algorithmus 4.2.4 *Berechnen eines B-Spline Fuzzy Reglers mit vorgeschalteter PCA.*

1. *Aufnehmen von Trainingsbeispielen.*
2. *Normalisierung der Eingabedaten, wenn es sich um Bilder handelt.*
3. *Abziehen des Mittelwertes von den Beispielen.*

4. *Wandeln der Eingabedaten in Vektoren und berechnen der Kovarianz- bzw. der impliziten Kovarianzmatrix.*
5. *Berechnen der Eigenwerte und -vektoren.*
6. *Wählen der geeigneten Anzahl von Eigenvektoren.*
7. *Projektion der Trainingsbeispiele in den Eigenraum.*
8. *Wahl der Parameter des B-Spline Reglers.*
9. *Training des B-Spline Reglers mit den projizierten Trainingsbeispielen.*

Die konkrete Durchführung der Punkte 1-7 ist anhand eines Beispiels in Kapitel C beschrieben.

4.3 Output Relevant Feature (ORF)

Die PCA stellt in erster Linie eine Dimensionsreduzierung dar. Es wird dabei aber nicht immer eine Korrelation zwischen den Eingangs- und den gewünschten Ausgangsdaten hergestellt. Anhand der Feinpositionierung eines Roboters mittels einer Handkamera über einem Bauteil ist dies einfach zu erläutern. Bei der Feinpositionierung muss sowohl die Position wie auch die Orientierung des Manipulators um die Annäherungsachse korrigiert werden. Zum Training wird der Roboter zunächst geeignet über dem später zu greifenden Bauteil positioniert. Anschießend wird der Roboter ausgelenkt und Bilder in verschiedenen Rotationsstellungen aufgezeichnet. Bei der Berechnung der *principal components* wird auffallen, dass die Information über die Translation des Bauteils im Kamerabild erhalten bleibt. Die Rotation des Bauteils wird sich nur schwer in der Projektion wiederfinden. Dies kommt dadurch zustande, dass nur die Varianz in den Daten betrachtet wird, unabhängig von den wirklich benötigten Informationen.

Eine Lösung besteht darin, Werte zu ermitteln, die eine Korrelation zwischen dem Eingangsraum und Ausgangsraum beinhalten. Diese Werte werden im folgenden als *Output Relevant Feature* (ORF) bezeichnet.

Die Berechnung solcher Werte wird in [Sch98b] vorgestellt, wobei das Verfahren der iterativen Berechnung der PCA ähnelt. Die Gewichtsvektoren werden aber nicht so modifiziert, dass die Varianz der Eingangsdaten entlang der Gewichtsvektoren maximiert wird, sondern dass der direkte Fehler zwischen Soll- und Ist-Wert minimal wird. Im Gegensatz zur PCA ist dies ein überwachtes Lernverfahren, da die Sollausgabe bei der Berechnung der ORF bekannt sein müssen. Dies ist in diesem Fall aber keine Einschränkung, da diese Werte für das Training des nachgeschalteten B-Spline Fuzzy Reglers auch zur Verfügung stehen müssen.

Die Lernregel 4.7 ändert sich daher wie folgt:

$$\Delta w_j = \eta(Y_S - V)\xi_j$$

mit Y_S als der Sollausgabe. Die Berechnung mehrerer Vektoren ist äquivalent zu der iterativen Berechnung der *principal components*.

Im Gegensatz zur PCA wird nur ein Projektionsvektor benötigt, da die Projektion schon die maximale Korrelation mit den Solldaten besitzt. Die Berechnung der ORF ist aber sehr rechenzeitintensiv.

4.4 Sensordatenfusion

Die Fusion von Sensordaten unterschiedlicher Sensoren unterteilt sich in 3 Phasen. In der ersten Phase wird ermittelt, welche Sensoren zur Lösung des Problems beitragen. Dies geschieht bei eindimensionalen Sensordaten durch die direkte Berechnung der Korrelation und bei mehrdimensionalen (z.B. bei Bildern) durch die Berechnung der *Output relevant Features*, bei der ebenfalls die Korrelation der projizierten Daten ermittelt wird. Unterschreitet die berechnete Korrelationen einen Schwellwert, so werden diese Sensordaten als nicht-relevant klassifiziert und auch nicht weiter berücksichtigt. Wird für die Reduktion von mehrdimensionalen Sensordaten die PCA verwendet, muss die Bewertung manuell erfolgen, da kein objektives Kriterium vor dem Training des B-Spline Fuzzy Reglers zur Verfügung steht.

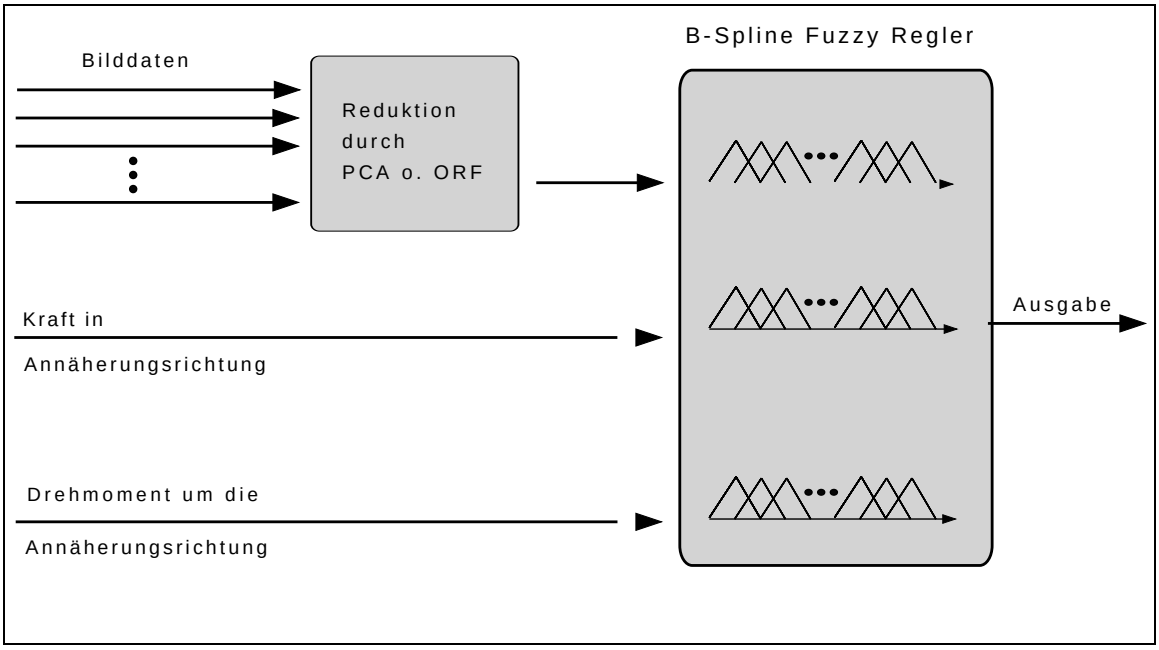


Abbildung 4.9: *Fusion mehrerer unterschiedlicher Sensordaten.*

In der zweiten Phase wird bei mehrdimensionalen Sensordaten eine Reduktion mittels PCA oder ORF durchgeführt. In der vorliegenden Arbeit wurde die Dimensionsreduzierung per ORF durchgeführt, da damit bessere Resultate im Vergleich zur PCA erzielt wurden und keine manuelle Bewertung erforderlich ist.

In der dritten und letzten Phase werden die relevanten Sensordaten, die unter Umständen in einen Unterraum projiziert worden sind, zu einem Vektor zusammengefasst und als Eingabe für einen B-Spline Fuzzy Regler verwendet. Abb. 4.9 zeigt den in [vCFZK00] verwendeten Ansatz.

Ein Zusammenfassen der Eingangsdaten zu einem einzigen Vektor und eine anschließende Reduzierung ist mit der PCA nicht möglich. Da unterschiedliche Sensordaten unterschiedliche Varianzen aufweisen, erhält man mit der PCA nicht die erwünschten Resultate. Es würden einige Sensoren aufgrund des Wertebereichs ihrer Daten bevorzugt. Bei ORF sollte eine solche Zusammenfassung theoretisch möglich sein, da dort eine entsprechende Gewichtung der Komponenten anhand der Korrelation ermittelt wird. Es hat sich ein solches Vorgehen aber in praktischen Versuchen als nicht gangbar erwiesen. Die Ursache liegt in der iterativen Berechnung, die darauf angewiesen ist, dass die Komponenten der Eingangsmuster im selben Wertebereich liegen. Gleichartige Sensordaten, wie z.B. Bilder, können und sollten vor einer Reduzierung zusammengefasst werden (siehe Abb. 4.12).

4.5 Beispiel: Erkennen der Orientierung einer Schraube

Das Beispiel ist [vCSZK99] entnommen[3]. Es geht hierbei um das Erkennen der Orientierung einer Schraube, um sie für den Schraubvorgang geeignet auszurichten. Der eine Roboter hält die Schraube, während der andere den Schraubwürfel hält (siehe Abb. 4.10). Die Szene wird durch zwei Kameras beobachtet; von oben und von der

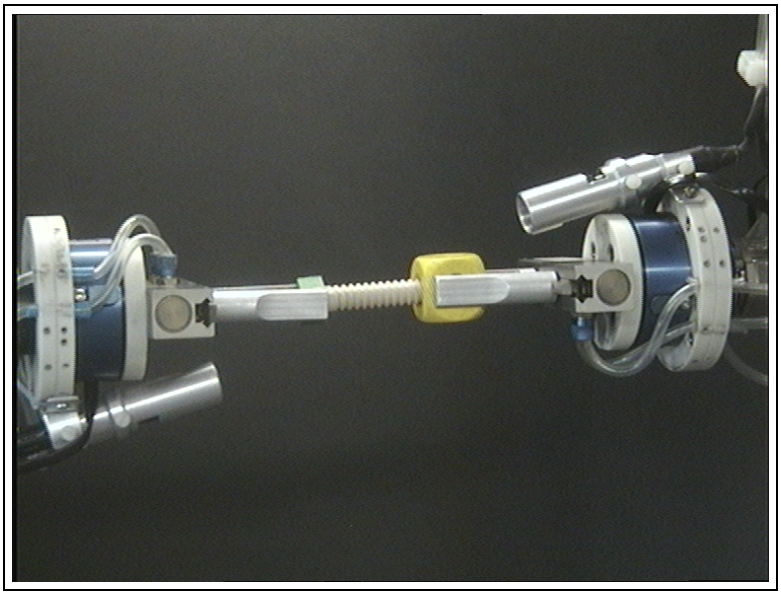


Abbildung 4.10: *Schraubvorgang mit einer langen Schraube.*

[3]Der entsprechende Quelltext zur Berechnung der PCA und der ORF wird in Kapitel C im Anhang erläutert.

Seite. Anhand der in Abb. 4.11 gezeigten exemplarischen Bildausschnitte, soll die Orientierung der Schraube erkannt werden.

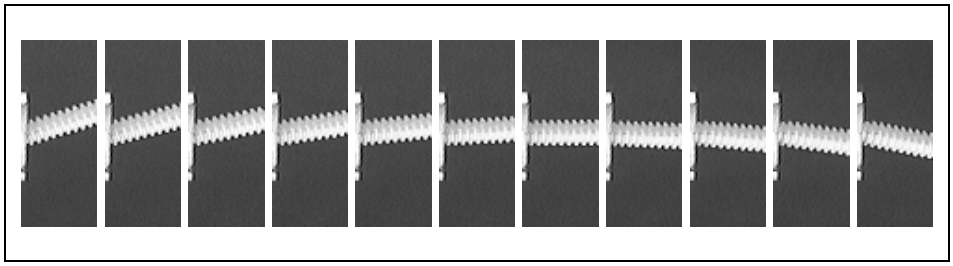


Abbildung 4.11: *Verschiedene Stellungen einer Schraube in einem Gewindeloch.*

Dimensionsreduzierung mit PCA

Vor Durchführung der PCA wurden beide Kameraansichten zusammengefasst (siehe Abb. 4.12). Sei $\vec{a}$ ein Bild der ersten Ansicht und $\vec{b}$ der zweiten, so ergibt sich das zusammengefasste Bild $\vec{c}$ als

$$\vec{c} = \begin{pmatrix} a_1 \\ \vdots \\ a_n \\ b_1 \\ \vdots \\ b_m \end{pmatrix}$$

mit $\vec{a} = (a_1, \ldots, a_n)^T$ und $\vec{b} = (b_1, \ldots, b_m)^T$. In der Regel ist es ausreichend, die Va-

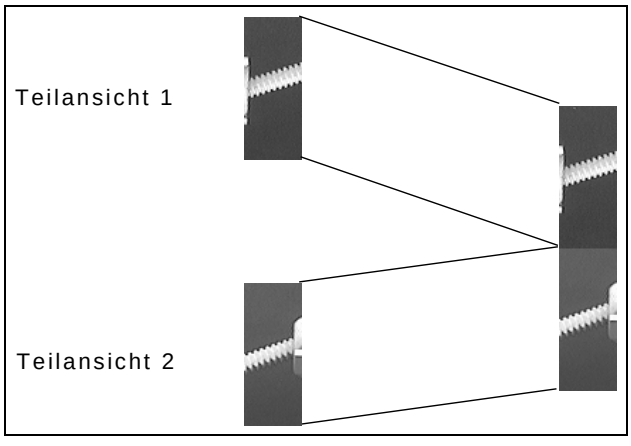
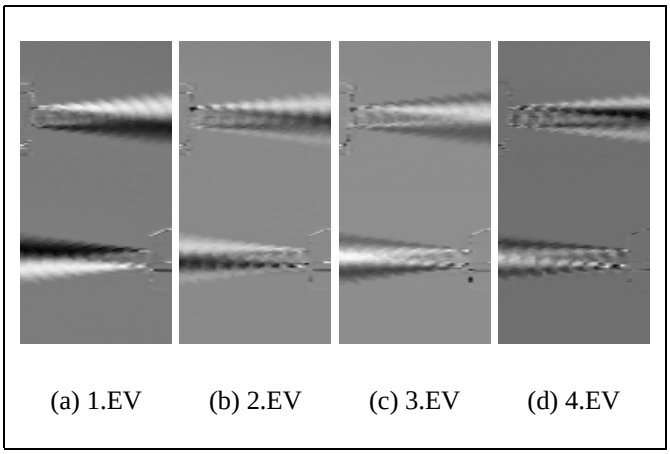


Abbildung 4.12: *Fusion zweier Ansichten ein und derselben Szene.*

rianz entlang der ersten drei bis vier Eigenvektoren zu berücksichtigen, daher genügt

es, die Projektion auf diese Vektoren zu betrachten. Über die Rücktransformation der Eigenvektoren in ein Bild lässt sich gut erkennen, welche Teile einer Ansicht stärker und welche Teile weniger stark berücksichtigt werden. Helle Pixel kennzeichnen eine hohe und dunkle eine niedrige Gewichtung. Es ist zu erkennen, dass gerade die Ränder des Gewindes bei einer Projektion auf die Eigenvektoren (EV) berücksichtigt werden (siehe Abb. 4.13).



(a) 1.EV (b) 2.EV (c) 3.EV (d) 4.EV

Abbildung 4.13: *Rücktransformation von Eigenvektoren in Graustufenbilder.*

Der Regler wurde insgesamt mit 362 Bilder trainiert und mit weiteren 362 Bilder getestet. Es wurden vier Eigenvektoren zur Dimensionierung verwendet, wobei die erste Dimension des reduzierten Bildes mit zehn, der zweite mit neun, die dritte mit acht und die vierte mit sieben B-Splines abgedeckt wurde. Tabelle 4.1 zeigt den größten Fehler und den größten ungünstigsten Fehler sowie den mittleren quadratischen Fehler für die verschiedenen Ansichten. Der ungünstigste Fehler bezeichnet im Gegensatz zum größten Fehler solche Fälle, in denen der Regler eine Bewegung veranlasst hätte, die zu einer Vergrößerung des Orientierungsfehlers der Schraube geführt hätten. Solche Fehler dürfen im realen Einsatz nicht auftreten. Die Testergebnisse zeigen die zu erwartenden Ergebnisse:

1. Die jeweiligen Ansichten der Szene liefern für die entsprechende Auslenkungsrichtung bessere Informationen. So ist die erste Ansicht zur Korrektur einer Auslenkung um die Normalenachse besser geeignet als die zweite Ansicht. Analog ist die zweite Ansicht besser zur Korrektur einer Auslenkung um die Schließachse geeignet.

2. Die Auswertung beider Ansichten für eine Ausrichtungsrichtung führt zu einem signifikant besseren Ergebnis. Insbesondere werden keine Fehlbewegungen initiiert.

	Größter Fehler	Ungünstigster Fehler	mittlerer quadratischer Fehler
N-Achse			
Ansicht 1	$11,14$	$7,63$	$8,81$
Ansicht 2	$29,85$	$29,85$	$57,91$
Ansicht 1 + 2	$3,14$	–	$0,69$
O-Achse			
Ansicht 1	$26,74$	$26,74$	$53,24$
Ansicht 2	$12,75$	$12,75$	$10,47$
Ansicht 1 + 2	$4,04$	–	$0,92$

Tabelle 4.1: Größter, ungünstigster und mittlerer quadratischer Fehler. Ist kein ungünstigster Fehler aufgeführt, so findet in dieser Sensorkonfiguration keine fehlervergrößernde Korrekturbewegung statt.

Dimensionsreduzierung mit ORF

Im Gegensatz zu Beispiel 4.5 werden hier die Bilddaten nicht über die PCA reduziert, sondern über ORF. Das gesamte Bild wird folglich auf einen Skalar projiziert. Abb. 4.14 zeigt die Rücktransformation der Gewichtsvektoren. Die Trainingsbeispiele sind dieselben.

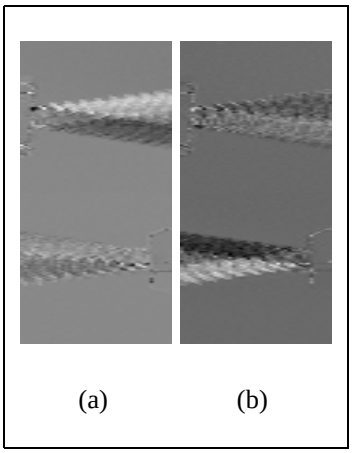


Abbildung 4.14: Rücktransformation des über ORF gewonnenen Projektionsvektors (a) für die Auslenkung um den Normalenvektor, (b) für die um den Schließvektor).

Abb. 4.14 zeigt die Unterschiede zwischen der PCA und ORF. Hier ist ebenfalls der Projektionsvektor in ein Graustufenbild zurückgewandelt worden und zeigt durch die Helligkeitsverteilung die Gewichtung eines Pixels im Bild an. Wie bei der Reduktion mit PCA zeigen die Ergebnisse, dass die jeweiligen Ansichten für die entsprechende

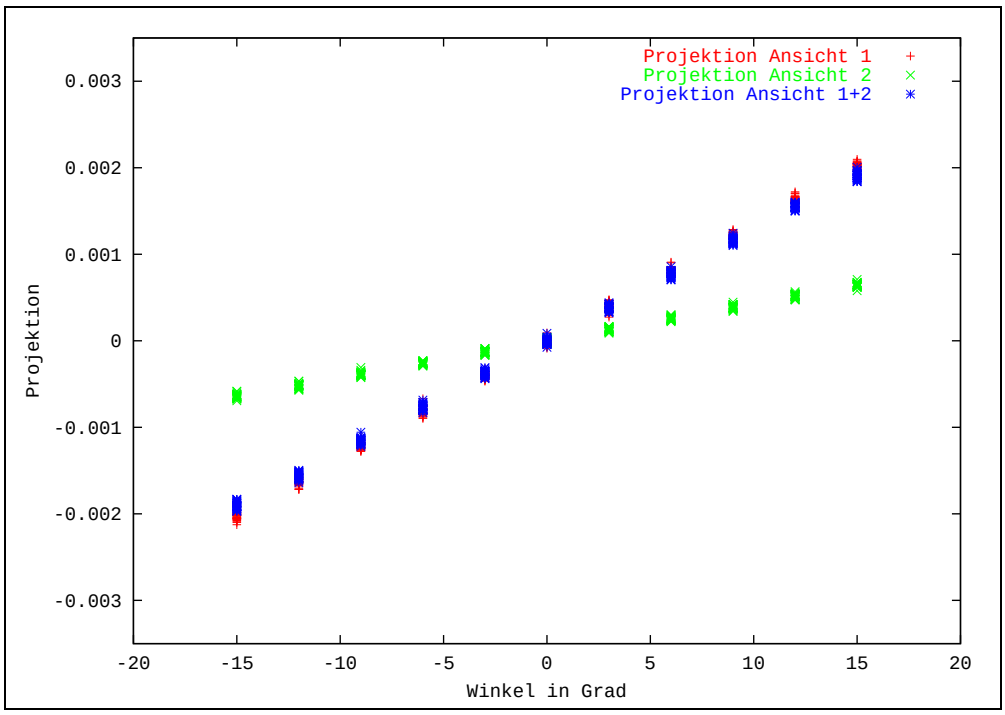


Abbildung 4.15: *Sollwinkel gegen Projektion der Bilder von der Draufsicht.*

Auslenkungsrichtung besser geeignet sind. Bei ORF wird dies schon während der Dimensionsreduzierung sichtbar. In Abb. 4.14(a) ist die obere Hälfte des Bildes und damit die erste Ansicht der Szene mehr berücksichtigt, in Abb. 4.14(b) die untere Hälfte. In Abb. 4.13 sind solche Unterschiede nicht zu erkennen. Dort werden die beiden Ansichten gleich stark gewichtet. Dies lässt sich auch in Abb. 4.15 und Abb. 4.16 erkennen. Dort ist der Sollwinkel gegen die Projektion aufgetragen. Die Projektion der fusionierten Sichten *Projektion 1+2* ist mit der Projektion der ersten (Abb. 4.15) bzw. der zweiten Ansicht (Abb. 4.16) fast identisch. Die Güte des entstandenen Reglers ist gleichzusetzen mit der des Reglers mit vorgeschalteter PCA. Im Gegensatz hierzu wurde aber das Bild stärker reduziert und der Eingang des B-Spline Fuzzy Reglers wurde von nur acht statt 10 B-Splines abgedeckt. Der Regler ist daher im Ressourcenverbrauch wesentlich günstiger.

Hinzunahme weiterer Sensoren

Wie in Kapitel 4.4 beschrieben, lassen sich nach diesem Prinzip auch Daten unterschiedlicher Sensoren fusionieren. Im Beispiel werden zuzüglich zu den Kameradaten auch die Sensordaten eines Kraftmomentensensors hinzugenommen [vCFZK00] (siehe Abb. 4.9). Um zu entscheiden, welche Daten des Sensors tatsächlich relevant sind, wird zuerst die Korrelation zwischen den Sollwerten und den Sensordaten berechnet. Die Sensordaten, bei denen der Betrag der Korrelation über 0.5 liegt, werden zur Bestim-

	Größter Fehler	Ungünstigster Fehler	mittlerer quadratischer Fehler
N-Achse			
Ansicht 1	$7,15$	–	$4,25$
Ansicht 2	20.25	$20,25$	$19,25$
Ansicht 1 + 2	$4,48$	–	$2,72$
O-Achse			
Ansicht 1	$9,39$	$9,394$	$8,71$
Ansicht 2	$5,48$	–	$3,62$
Ansicht 1 + 2	$4,08$	–	$2,69$

Tabelle 4.2: Größter, ungünstigster und mittlerer quadratischer Fehler bei der Dimensionsreduzierung durch ORF. Ist kein ungünstigster Fehler aufgeführt, so findet in dieser Sensorkonfiguration keine fehlervergrößernde Korrekturbewegung statt.

	Größter Fehler	Ungünstigster Fehler	mittlerer quadratischer Fehler
N-Achse			
Ansicht 1	$5,73$	–	$1,98$
Ansicht 1 + 2	$8,22$	–	$1,91$
O-Achse			
Ansicht 2	$6,19$	–	$2,85$
Ansicht 1 + 2	$6,14$	–	$2,63$

Tabelle 4.3: Größter, ungünstigster und mittlerer quadratischer Fehler bei der Dimensionsreduzierung durch PCA mit Hinzunahme von Kraftmomentendaten. Ist kein ungünstigster Fehler aufgeführt, so findet in dieser Sensorkonfiguration keine fehlervergrößernde Korrekturbewegung statt.

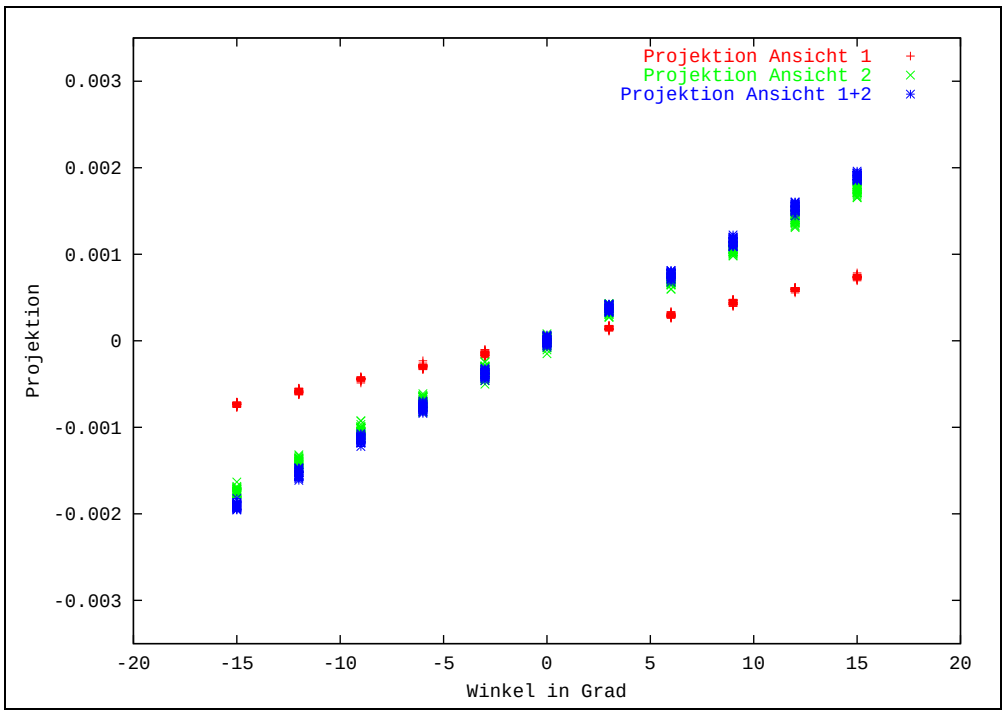


Abbildung 4.16: *Sollwinkel gegen Projektion der Bilder von der Seitansicht.*

mung der Orientierung herangezogen. Tabelle 4.3 zeigt, dass durch die Hinzunahme der Daten des Kraftmomentensensors die Erkennungsergebnisse bei nur einer Ansicht deutlich gesteigert werden können. Sie zeigt aber ebenfalls, dass eine Hinzunahme von weiteren Daten nicht in jedem Fall zu einer Verbesserung der Erkennungsleistung führt. Ist die Erkennungsleistung schon sehr gut, wie bei der Fusion der beiden Ansichten, stellt die Hinzunahme von weiteren Sensordaten unter Umständen nur eine Erhöhung des Eingangsrauschens dar, welches bei gleichbleibender Anzahl der Trainingsmuster zu einer Verschlechterung der Erkennung führt. Bei einer Reduktion der Bilddaten mit ORF ist der Effekt zu erkennen. Hier ist die Korrelation der dimensionsreduzierten Bilddaten mit den Sollwerten schon sehr hoch.

4.6 Zusammenfassung

Ein B-Spline Fuzzy Regler eignet sich nicht nur für die Kraftmomentenregelung, sondern kann auch für die Sensordatenfusion eingesetzt werden. Im Gegensatz zu den sonst üblichen Fusionsansätzen wird kein Systemmodell benötigt. Es werden die zur Verfügung stehenden Sensordaten als Eingabe für den Regler genommen, ohne eine Vorverarbeitung durchzuführen. Um den Ressourcenbedarf zu beschränken, müssen hochdimensionale Sensordaten in ihrer Dimension reduziert werden. Dies geschieht über die PCA oder ORF. Bei der PCA wird zunächst die Korrelation der Eingangsda-

	Größter Fehler	Ungünstigster Fehler	mittlerer quadratischer Fehler
N-Achse			
Ansicht 1	$6,29$	$6,29$	$3,74$
Ansicht 1 + 2	$5,72$	$5,72$	$1,81$
O-Achse			
Ansicht 2	$9,61$	$9,35$	$4,98$
Ansicht 1 + 2	$7,98$	–	$3,31$

Tabelle 4.4: Größter, ungünstigster und mittlerer quadratischer Fehler bei der Dimensionsreduzierung durch ORF und Hinzunahme von Kraftmomentendaten. Ist kein ungünstigster Fehler aufgeführt, so findet in dieser Sensorkonfiguration keine fehlervergrößernde Korrekturbewegung statt.

ten durch den Übergang auf die principal components auf Null gebracht und anschließend die Varianz der *principal components* zur Reduktion herangezogen. Bei den ORF wird die Korrelation zwischen Projektion und Solldaten maximiert. Die Reduktion mittels ORF ist vorzuziehen, da die Eingangsdaten auf nur einen Skalar reduziert werden und gleichzeitig eine Bewertung über die Relevanz des Sensors für das gegebene Problem gemacht werden kann. Die Reduktion mittels PCA ist dagegen vom Rechenaufwand günstiger.

Die Berechnung eines Reglers besteht aus drei Schritten:

1. Auswahl der signifikanten Sensordaten.
2. Reduktion und unter Umständen Fusion hochdimensionaler Sensordaten.
3. Training des B-Spline Fuzzy Reglers.

Kapitel 5

Diskrete Ereignisabläufe

Wie im ersten Kapitel erläutert, ist die Entwicklung eines Lernverfahren zur Montage ein Ziel dieser Arbeit. Im Gegensatz zu der in [Mos00] vorgestellen Ansatzes, bei dem die Montagesequenzen durch die Dekomposition der zu montierenden Baugruppe aufgestellt wird, beschreibt ein Instrukteur über sprachliche Anweisungen den Montagevorgang, welcher durch das Montagesystem online durchgeführt wird. Die Anweisungen werden entsprechend ihrer zeitlichen Reihenfolge gespeichert. Ist die Instruktion der Montage abgeschlossen, so werden die Montageanweisungen dauerhaft abgelegt und können zu einem späteren Zeitpunkt wiederholt werden. Für ein einfaches Aggregat, z.B. ein Leitwerk ohne Ausrichten der Leiste, wird ein Montagedurchgang genügen. Für komplexere Aggregate werden dem System mehrere und unterschiedliche Beispiele instruiert werden müssen. Hat das Montagesystem genügend Beispiele für ein spezielles Aggregat gesammelt, kann aus diesen Beispielen und den in ihnen enthaltenen Daten weiteres Montagewissen abgeleitet werden.

Die Anweisungen des Instrukteurs an das System umfassen nicht nur direkte Bewegungsanweisungen[1] an die Manipulatoren, sondern hauptsächlich Instruktionen wie: *Greife eine gelbe Schraube* oder *Schraube die Schraube in den Würfel* . Dies setzt voraus, dass das Montagesystem a priori Montagekenntnisse besitzt, um Äußerungen wie *Schrauben* und *Greifen* mit einer begrenzten Autonomie umsetzen zu können. Das realisierte System kann im ungelernten Zustand folgende Anweisungen durchführen:

Greifen: Das Greifen eines Objektes.

Stecken: Das Stecken einer Leiste oder eines Schraubwürfels auf eine Schraube.

Schrauben: Das Schrauben einer Schraube in einen Schraubwürfel oder in eine Rautenmutter.

Ablegen: Das Ablegen eines Objektes oder Aggregates auf dem Montagetisch.

[1] Z.B.: “Fahre Manipulator 1 an die Position xyz”.

Zu diesen komplexen Operationen kann das Montagesystem folgende direkten Bewegungsanweisungen (motorische bzw. sensormotorische Operationen) verarbeiten: *Einstellen einer definierten Kraft am Endeffektor, Wechsel der Endeffektororientierung, Öffnen und Schließen der Greifer, Positionierung über einem Bauteil, Bewegungen des Roboters im Euklidischen und im Konfigurationsraum, Stecken eines Gegenstandes in ein Loch, kraftkontrolliertes Rotieren des Endeffektors, Schraubvorgang mit einem oder zwei Robotern, kraftkontrollierte Suche eines Loches oder Gewindes in einem Bauteil, Abspeichern einer Position.*

Das Montagesystem besitzt also eine Auswahl von direkten und komplexen Anweisungen, mit deren Hilfe der Instrukteur das Montagesystem anleitet. Im Unterschied zu den in Kapitel 2.2.1 vorgestellen Montageoperationen führen die durch den Instrukteur veranlassten Operationen des Montagesystems nicht immer eine Montage im Sinne von Kapitel 2.2.1 durch. So sind z.B. die Anweisungen zum greifen von Bauteilen keine eigentlichen Montageoperationen. In [Mos00] werden diese Operationen als Aktionsprimitive deklariert, aus denen die Montageoperationen oder auch Roboteraufgaben zusammengesetzt werden.

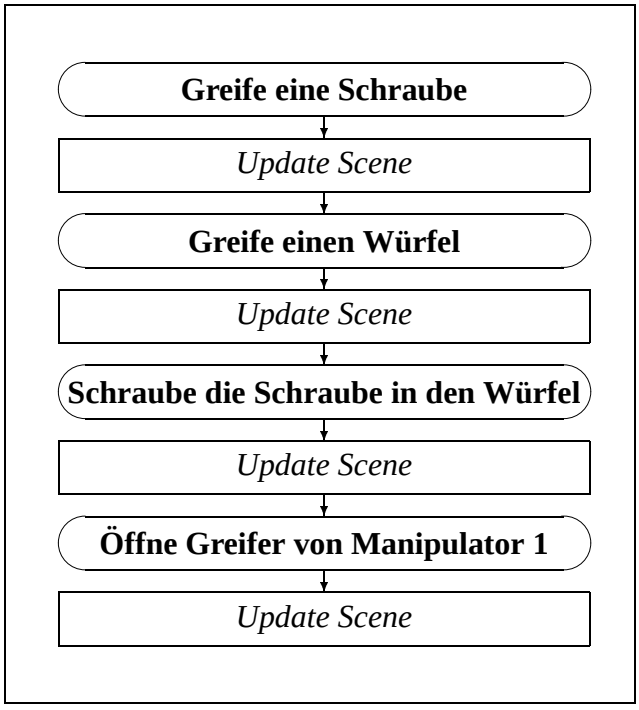


Abbildung 5.1: *Flussdiagramm einer Beispielsequenz.*

5.1 Generierung und Repräsentation

Zur Repräsentation der Montagefolge bedient sich das System der in Kapitel 3.3.2 vorgestellten Sequenzen, wobei $\pi(i)$ mit dem in Kapitel 2.2.1 eingeführten τ_i korrespondiert. Darüber hinaus besitzt das Montagesystem eine interne Repräsentation der Montageszene. Wird dem System eine Anweisung gegeben, so werden nacheinander folgende Schritte durchgeführt:

1. Prüfen, ob die Anweisung durchgeführt werden kann. Das System versucht im Rahmen seiner Möglichkeiten zu prüfen, ob eine Aktion durchführbar bzw. mit dem aktuelle Kontext stimmig ist. Eine Schrauboperation mit zwei Schrauben ist durchführbar, aber nicht sinnvoll und wird daher abgewiesen. Ist die Anweisung eine direkte Bewegungsanweisung, so ist das System nicht in der Lage zu beurteilen, ob dies eine sinnvolle Aktion ist und führt sie ohne Prüfung aus.

2. Ist die Operation **nicht** fehlgeschlagen, so wird sie als Anweisung inkl. des vorher gespeicherten Roboterzustandes in einer Sequenz abgelegt.

3. Als nächster Schritt wird ein spezielles Modul aufgerufen, das aufgrund der durchgeführten Operation die interne Szenenrepräsentation aktualisiert. War die Operation eine direkte Bewegungsanweisung an den Roboter, so ist dies nicht immer exakt möglich. Auch dieser Aufruf wird als eine Sequenzanweisung gespeichert (der Befehl `Update Scene` in Abb. 5.1).

Das Ergebnis ist eine durch den Instrukteur instruierte und als Sequenz abgespeicherte und auch durchgeführte Montagesequenz, die in Form eines *OPERA*-Skriptes abgelegt wird. Eine einzelne Anweisung des Instrukteurs wird dabei als zwei Instruktionen einer Sequenz abgelegt. Die erste Anweisung führt die Montageoperation real (Abb. 5.1 (fett)), die zweite virtuell durch (Abb. 5.1 (kursiv)). Das Montagesystem ist somit in der Lage, eine gelernte Montagesequenz simulativ durchzuführen, indem es nur jede zweite Anweisung ausführt. Abb. 5.1 zeigt eine solche Beispielsequenz. Es soll erst eine Schraube und ein Schraubwürfel gegriffen und dann zusammengeschraubt werden. Zum Beginn eines Lerndurchgangs wird der aktuelle Roboterzustand (was halten beide Roboter in den Händen) zusätzlich zur Sequenz gespeichert. Das Montagesystem kann anhand dieses Zustandes beurteilen, in welchem Kontext diese gelernte Sequenz später ausgeführt werden darf.

Eine so entstandene Sequenz kann auch in eine in Kapitel 2.2 vorgestellten Repräsentationen überführt werden (z.B. Abb. 5.2). Es ist aber zu beachten, dass die in Kapitel 2.2 vorgestellten Repräsentationen üblicherweise dafür verwendet werden, alle durchführbaren Montagesequenzen zu beschreiben, während in dieser Arbeit nur Sequenzen betrachtet werden, die durch den Instrukteur instruiert wurden. Ein daraus gebildeter Montagegraph ist nicht notwendigerweise vollständig. Ebenfalls ist zu beachten, dass die in dieser Arbeit verwendeten Sequenzen nicht nur eine Beschreibung enthalten, in welcher Reihenfolge ein Aggregat zusammengebaut werden muss, sondern

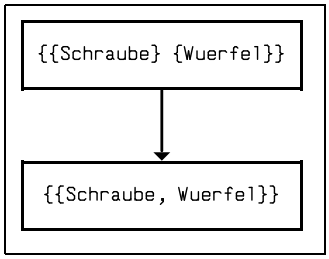


Abbildung 5.2: *Beispielsequenz aus Abb. 5.1 als direkter Montagegraph.*

sie enthalten insbesondere **wie** die Montage durchgeführt werden muss. Diese Information ist in den Repräsentationen aus Kapitel 2.2 nicht enthalten. Diese kommen üblicherweise dadurch zustande, dass die zu montierende Aggregate als Graphrepräsentation vorliegen und durch deren (virtuelle) Demontage der Graph aufgebaut wird. Dies findet ohne direkten Bezug auf die später verwendeten Manipulatoren statt. Erst nachgelagerte Operationen auf dem Montagegraph erzeugen eine zu verwendende Sequenz von Roboteroperationen.

5.1.1 Interne Repräsentation des Roboterzustandes

Würde nur die Abfolge der verschiedenen Roboteroperationen als eine Sequenz abgespeichert werden, so wären die Möglichkeiten des Systems, diese Sequenz weiter zu verarbeiten, sehr eingeschränkt. Weitergehende Operationen als nur das einfache Wiederholen der Handlung wären nicht möglich, da das Montagesystem keinerlei Informationen besitzt, was seine Handlungen bewirken. Der direkte Ansatz um Handlung und dessen Wirkung herauszufinden, wäre die Beobachtung der Handlung und der Veränderungen, die diese Handlung in der Szene bewirkt. Dies setzt aber eine leistungsstarke Bildverarbeitung voraus, die nicht nur Bauteile und Aggregate auf dem Montagetisch, sondern auch in den Greifern der Manipulatoren erkennt. Ein solches Bildverarbeitungssystem stand nicht zur Verfügung und wurde in der vorliegenden Arbeit durch die Simulation der Handlung ersetzt. Ausgehend von dem Wissen über die Semantik der verfügbaren Anweisungen, wird die reale Handlung virtuell nachvollzogen, so dass das Montagesystem zu jedem Zeitpunkt eine interne Repräsentation der Szene besitzt. Diese beinhaltet nicht nur die auf dem Montagetisch liegenden Bauteile sondern auch den Greiferinhalt.

Solange sich die Simulation der Handlungsanweisungen auf *Greifen, Schrauben, Stecken, Ablegen*, deren Semantik bekannt ist, beschränkt, ist es möglich die interne Repräsentation mit der Realität weitestgehend identisch zu halten. Schwierigkeiten treten dann auf, wenn direkte Bewegungsanweisungen wie z.B Relativbewegungen an die Manipulatoren gegeben werden. Deren Auswirkungen sind nicht leicht zu rekonstruieren, wenn Bauteile durch die Operation bewegt sein sollten.

Die interne Repräsentation beruht auf einem Objektmodell der *Baufix*-Bauteile das während der Arbeit zu [Fer01] entstanden ist. Es ist ein erweitertes relationales Bau-

gruppenmodell, das die Topologie der Baugruppe repräsentiert. Es modelliert zur Zeit folgende Bauteile:

- Schrauben,
- Schraubwürfel,
- Leisten,
- Rautenmuttern und
- die entsprechenden Verbindungen zwischen diesen Bauteilen.

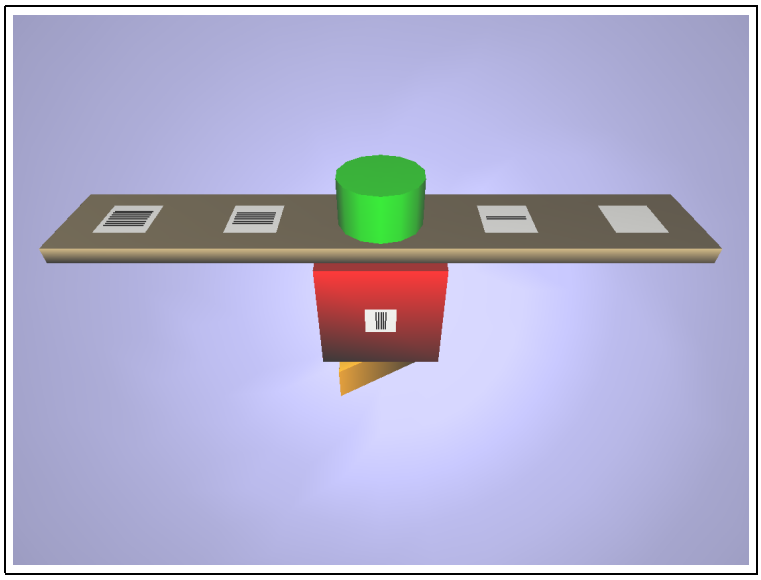


Abbildung 5.3: *Darstellung der modellierten Bauteile.*

Abb. 5.3 zeigt eine Darstellung der modellierten Bauteile (Leiste, Schraubwürfel und Rautenmutter aufeinandergeschraubt). Mit Hilfe dieser Modellierung kann nicht nur eine interne Repräsentation der gegriffenen Bauteile und Aggregate erstellt, sondern auch einfache Plausibilitätsabfragen über die Durchführbarkeit von Operationen getätigt werden. Durch die Modellierung ist das Montagesystem später in der Lage zu entscheiden, ob bestimmte (Teil-)Aggregate schon einmal gebaut worden sind und ob es sich schon innerhalb eines bekannten Montageplans befindet.

In der Realisierung der vorliegenden Arbeit werden sowohl die Bauteile und Aggregate auf dem Montagetisch, als auch die in den Greifern modelliert. In Abbildung 5.4 ist ein Beispiel einer interne Repräsentation visualisiert. Sie zeigt verschiedene Bauteile auf dem Tisch sowie eine gegriffene Schraube und Leiste. Die Manipulatoren werden selbst nicht dargestellt.

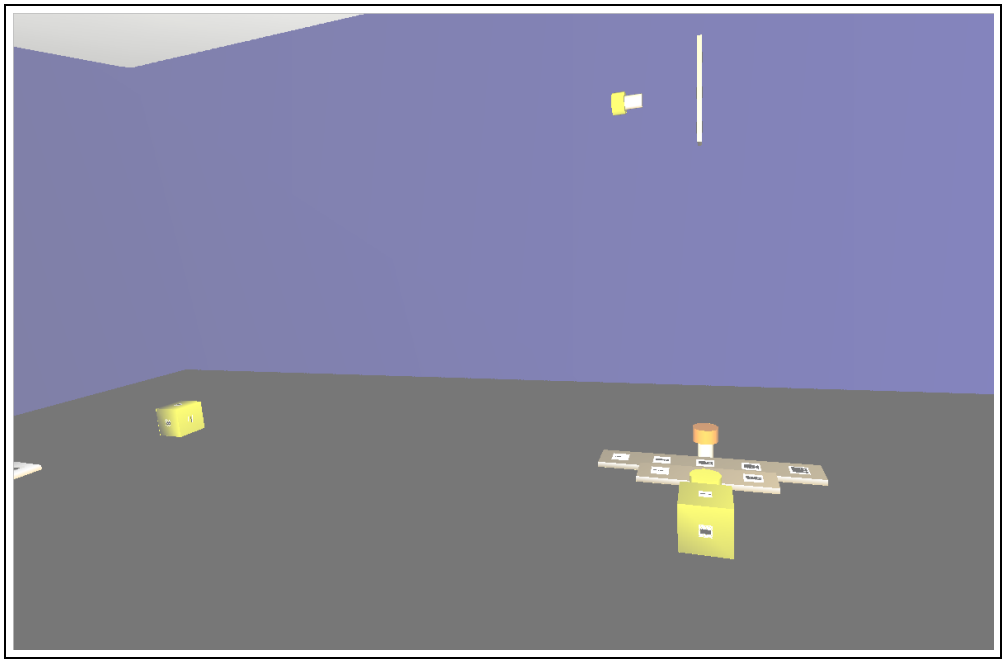


Abbildung 5.4: *Beispielvisualisierung eines internen Zustandes. Die Abbildung zeigt mehrere auf dem Montagetisch liegende Bauteile und eine Leiste und eine Schraube, die von den (nicht dargestellten) Robotern gehalten werden.*

5.1.2 Sensortrajektorie

Im Regelfall wird der Instrukteur mehrere Beispiele für einen Montagevorgang geben, die vom System aufgezeichnet werden. Da für den Bau ein und desselben Aggregates womöglich unterschiedliche Handlungsstränge gezeigt werden, muss das Montagesystem Informationen erhalten, woran es die unterschiedlichen Montagefolgen später unterscheidet. Die einzigen Informationen, die das System zusätzlich akquirieren kann, sind Sensordaten.

Während des Lernvorgangs speichert das Montagesystem alle ihm zur Verfügung stehenden Sensordaten zusätzlich zu den Anweisungen des Instrukteurs auf. Somit besteht die gespeicherte Sequenz nicht nur aus Anweisungen an den Roboter, sondern zu jeder Anweisungen sind zusätzlich die Sensordaten, die zum Zeitpunkt **vor** der Ausführung der jeweiligen Operation gültig waren, abgelegt worden.

Um zu ermitteln, welche Sensoren und damit welche Sensordaten zur Verfügung stehen, bedient sich das Montagesystem des in Kapitel 3.3.4 vorgestellten Mechanismus von *OPERA*, den Sensorzugang über Module zu realisieren. Somit müssen nur die Daten von allen Modulen, die Sensorinformationen zur Verfügung stellen, angefordert werden. Unterschiedliche Sensoren können so einfach ins Lernverfahren integriert werden.

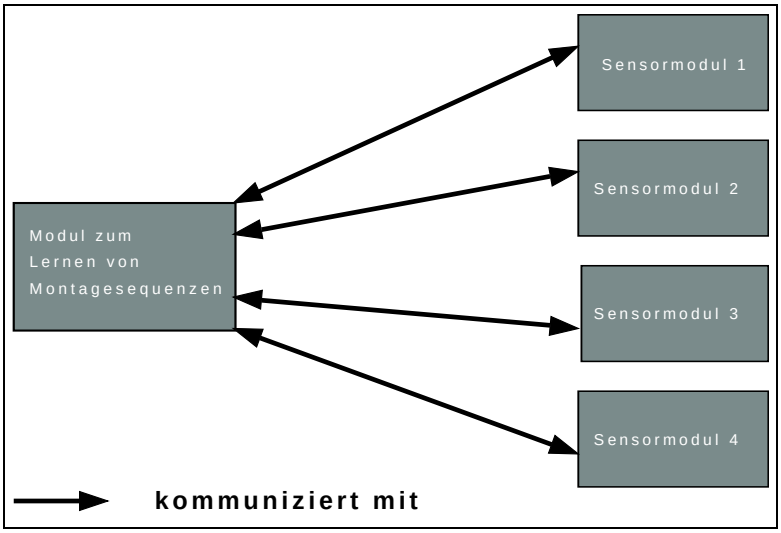


Abbildung 5.5: *Sensordatenaquirierung.*

5.2 Generierung bei Ausnahmen

Eine wichtige Frage ist: Was passiert, wenn der Instrukteur während des Lernvorganges eine Operation abbricht oder wenn das System von sich aus einen Fehler entdeckt? Das Ziel ist dabei, in einen früheren Zustand des Montagesystems überzugehen, von dem aus stets die Montage und der Lernvorgang weitergeführt werden können. Dabei sind verschiedene Fälle zu berücksichtigen:

- Die Montagehandlung wird während der Ausführung einer komplexen Anweisung (z.B. Schrauben) unterbrochen.
- Die Montagehandlung wird während der Ausführung einer direkten Bewegungsanweisung unterbrochen.
- Während das Montagesystem eine ihm bekannte und gelernte Montagefolge abarbeitet, unterbricht der Instrukteur die Montage.

Bei jedem dieser möglichen Fälle gibt es verschiedene Probleme zu beachten. Entdeckt das Montagesystem selbst einen Fehler, so ist dies weniger kritisch, solange nicht die Szene entscheidend modifiziert worden ist. Ist z.B. ein Schraubvorgang nicht möglich, da der Gewindeeinschnitt nicht gefunden wurde, so wird dies vom System erkannt und die Anweisung wird nicht abgelegt. Der Systemzustand hat sich nicht entscheidend geändert. Ist aber der Schraubvorgang an einem Zustand unterbrochen worden, aus dem das Montagesystem ohne aktive Hilfe des Instrukteurs nicht herauskommt (z.B. mechanische Defekte), so muss in den meisten Fällen der Lernvorgang abgebrochen werden.

Um einen Abbruch zu umgehen, ist das Führen eines Fehlerdialogs notwendig. Dem Montagesystem muss mitgeteilt werden, dass ein Fehlerzustand vorliegt und dass die

folgenden Anweisungen aus diesem Fehlerzustand wieder herausführen. Ziel ist es auch hier, einen früheren Zustand im Montageprozess zu erreichen, so dass die Montage von dort aus fortgeführt werden kann. Problematisch ist der Übergang vom Fehlerzustand in den Zustand des normalen Lernvorgangs. Sowohl für das Montagesystem als auch für den Instrukteur ist es schwierig, sicher zu beurteilen, ob der Zustand vor dem Auftreten der Ausnahme noch regulär war.

Die Unterbrechung während einer komplexen Anweisung ist die am einfachsten zu behandelnde Unterbrechung. Da das Montagesystem die Semantik dieser Operationen kennt, weiß es auch ob und wie es in den vorherigen Zustand zurückkommt. Es ist dem System möglich Demontageoperationen selbständig durchzuführen. Stellt z.B. der Instrukteur während eines Schraubvorgangs fest, dass diese Operation nicht richtig ist, so kann diese rückgängig gemacht werden.

Wurde dagegen eine direkte Anweisung für einen speziellen Manipulator gegeben, so sind die Auswirkungen dieser Anweisung für das Montagesystem entweder nicht bekannt oder es fehlt das Wissen, wie und ob die Auswirkung rückgängig gemacht werden kann. Eine selbständige Rückführung in einen früheren Zustand ist nicht immer möglich.

Bei der Ausführungsunterbrechung einer Montagesequenz treten dieselben Schwierigkeiten auf wie bei der Unterbrechung einer direkten Anweisung an den Manipulator. Eine Montagesequenz besteht aus einer Verkettung von komplexen und direkten Anweisungen. Das Montagesystem kann daher nicht in jedem Fall wissen, wie der Vorgang rückgängig gemacht werden kann.

Lösungsansatz

Beschränkt man sich auf komplexe Anweisungen, wie Greifen, Schrauben, Stecken, Ablegen, und setzt man voraus, dass direkte Anweisungen die Szene nicht entscheidend manipulieren, außer sie lassen sich durch die Einnahme einer vorherigen Konfiguration des Manipulators rückgängig machen, so bietet sich der folgende Ansatz an.

Wie in Kapitel 3.3.3 beschrieben besitzt *OPERA* mehrere Möglichkeiten mit Ausnahmen umzugehen. Eine dieser Möglichkeiten besteht darin, im Ablauf einer Sequenz zurück zu springen und dort Anweisungen auszuführen, die bei einer normalen Abarbeitung nicht ausgeführt würden (*Recover*-Abschnitt). Abb. 5.6 zeigt eine Beispielsequenz mit einem solchen Abschnitt. Bei normaler Ausführung dieser Sequenz würde die (fett) markierte Instruktion nicht ausgeführt. Erst wenn die letzten beiden Anweisungen der Sequenz einen entsprechenden Fehlercode zurückliefern, wird diese Instruktion ausgeführt (angedeutet durch die gestrichelten Kanten). In diesem Beispiel würde der erste Manipulator zurückgezogen und die Ausführung der Sequenz beendet.

Des Weiteren wird davon ausgegangen, dass beim Auftreten eines Ausnahmezustandes während der Ausführung einer komplexen Anweisung, das Montagesystem weiß, wie es in den Zustand vor der Anweisung zurückgelangt. Damit verbleibt die Problematik, wie man mit Ausnahmen bzw. der Stopp-Anweisung während des Ausführens

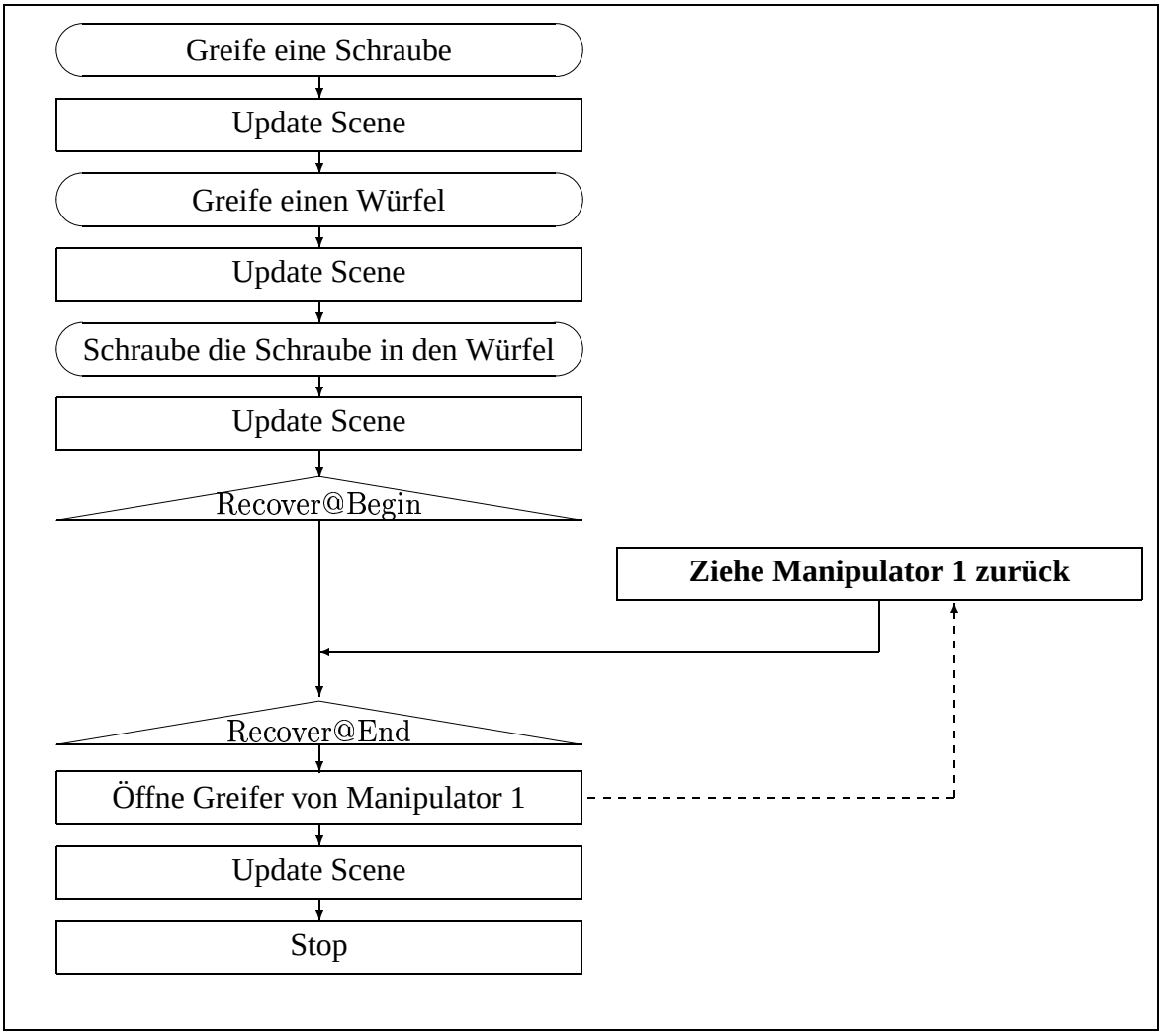


Abbildung 5.6: *Flussdiagramm einer Beispielsequenz mit* Recover-*Abschnitt.*

einer direkten Anweisung umgeht. Geht man von der oben gemachten Voraussetzung aus, dass nur die Position der Manipulatoren im Konfigurationsraum und der Zustand der Greifer restauriert werden müssen, so müssen vor jede Anweisung weitere Instruktionen eingefügt werden, die die Position und den Zustand speichern. Dies wird in Abb. 5.7 gezeigt. Mit dem Einfügen dieser Anweisungen ist es sichergestellt, dass die Position und der Greiferzustand der Manipulatoren, vor jeder Operation gesichert werden. Zusätzlich muss noch der oben beschriebene *Recover*-Abschnitt eingefügt werden (Abb. 5.8). Wird während des Lernvorganges vom Instrukteur eine direkte Manipulatoranweisung gegeben, so wird diese in eine Sequenz (Abb. 5.8) umgesetzt, die von *OPERA* ausgeführt wird. Tritt nun während der Ausführung der eigentlichen Anweisung

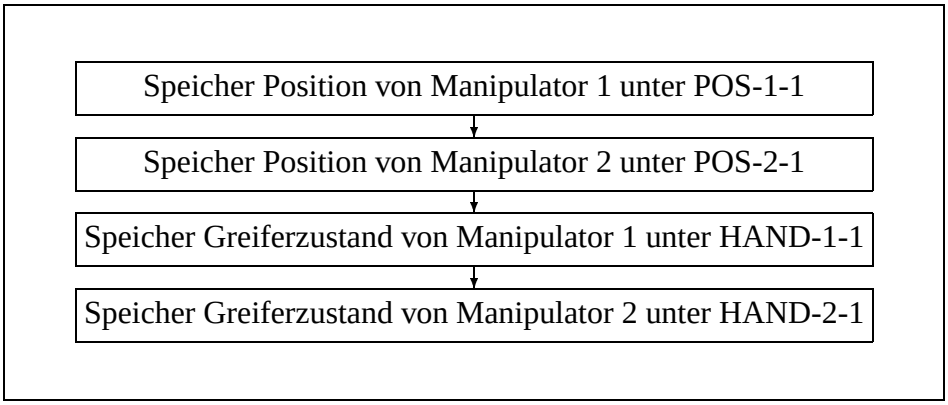


Abbildung 5.7: *Flussdiagramm einer Beispielsequenz zum Speichern der Position und des Greiferzustandes.*

an den Manipulator eine Ausnahme[2] auf, kann der *Recover*-Abschnitt angesprungen und der ursprüngliche Zustand der Roboter wiederhergestellt werden.

Sollen mehrere Anweisungen rückgängig gemacht werden, so sind folgende Punkte zu beachten:

- Die Manipulatorzustände müssen in jeweils verschiedenen Variablen gespeichert werden.
- Jeder *Recover*-Abschnitt muss mit einer eindeutigen Sprungmarke versehen werden.
- Im *Recover*-Abschnitt muss eine Abfrage stattfinden, ob weitere Ausführungsschritte zurückgegangen werden soll.

In der Praxis läuft dieser Lösungsansatz darauf hinaus, dass sämtliche Operationen Wissen darüber besitzen, wie die von ihnen durchgeführten Operationen und Auswirkungen dieser Aktionen rückgängig gemacht werden können, was unter Umständen die Verwendbarkeit dieser Operationen einschränkt.

Der viel flexiblere Ansatz wäre, dass der Instrukteur, ebenfalls durch Anweisungen, das Montagesystem in einen früheren Zustand zurückführt. Ist der angestrebte frühere Zustand erreicht, muss dieses dem Montagesystem mitgeteilt werden, sollte es dies nicht selbständig erkennen. Wie schon oben erwähnt, ist es aber für beide Parteien nicht einfach festzustellen, wann und ob ein früherer Zustand erreicht worden ist. Das Montagesystem besitzt gegenüber dem Instrukteur nur eine sehr eingeschränkte Sicht der Szene und muss sich darauf verlassen, dass die interne Repräsentation hinreichend genau die Realität approximiert. Entscheidet der Instrukteur, ob ein frühere Zustand erreicht worden ist, riskiert er, dass das Montagesystem unterschiedliche Situationen als identisch einstuft.

[2] Dies wird in der Regel ein STOPP-Kommando des Instrukteurs sein.

5.3 Zusammenfassung

Das Montagesystem lernt die Montage von Aggregaten, indem der Instrukteur dem System Schritt für Schritt über sprachliche Anweisungen die durchzuführende Handlung angibt. Diese Anweisungen sind entweder direkt Bewegungsanweisung an die Manipulatoren oder komplexe Montageanweisungen. Die einzelnen Instruktionen werden zusammen mit dem **vor** der Ausführung gültigen Sensorzustand in einer Sequenz gespeichert. Zusätzlich wird eine Anweisung generiert, die die interne Szenenrepräsentation entsprechend der Anweisung des Instrukteurs mit der realen Szene abgleicht. Die durch den Lernvorgang entstandene Sequenz ist ein Beispiel für den Bau des gewünschten Aggregates. Je mehr unterschiedliche Möglichkeiten für den Bau des Aggregates existieren, um so mehr Beispielsequenzen werden benötigt. Im Gegensatz zu klassischen Repräsentationen von Montagesequenzen wird nicht nur die Reihenfolge der zu montierenden Teilaggregate aufgezeichnet, sondern auch die konkreten Operationen der Manipulatoren.

Für den Fall einer Unterbrechung einer Handlung durch Eintreten eines Fehlers oder Intervention, versucht das Montagesystem in Kooperation mit den Instrukteur einen Systemzustand vor dem Beginn des Fehlers einzunehmen. Es wird die Einschränkung gemacht, dass direkte Anweisungen an den Manipulator durch Herstellen eines früheren Roboterzustandes rückgängig gemacht werden können. Komplexe Anweisungen sind von sich aus in der Lage in einen früheren Zustand zurückzukehren. Zur Realisierung wird auf Fehlerbearbeitungsmechanismen von *OPERA* zurückgegriffen.

Dieser Ansatz unterscheidet sich wesentlich von den üblichen Verfahren eine Montagesequenz zu generieren, da die zu montierende Baugruppe zu Beginn der Montage dem System nicht bekannt ist. Für die Ansätze zu *assembly by dissassembly* ist dies aber eine wesentliche Voraussetzung, um einen Montageplan aufstellen zu können.

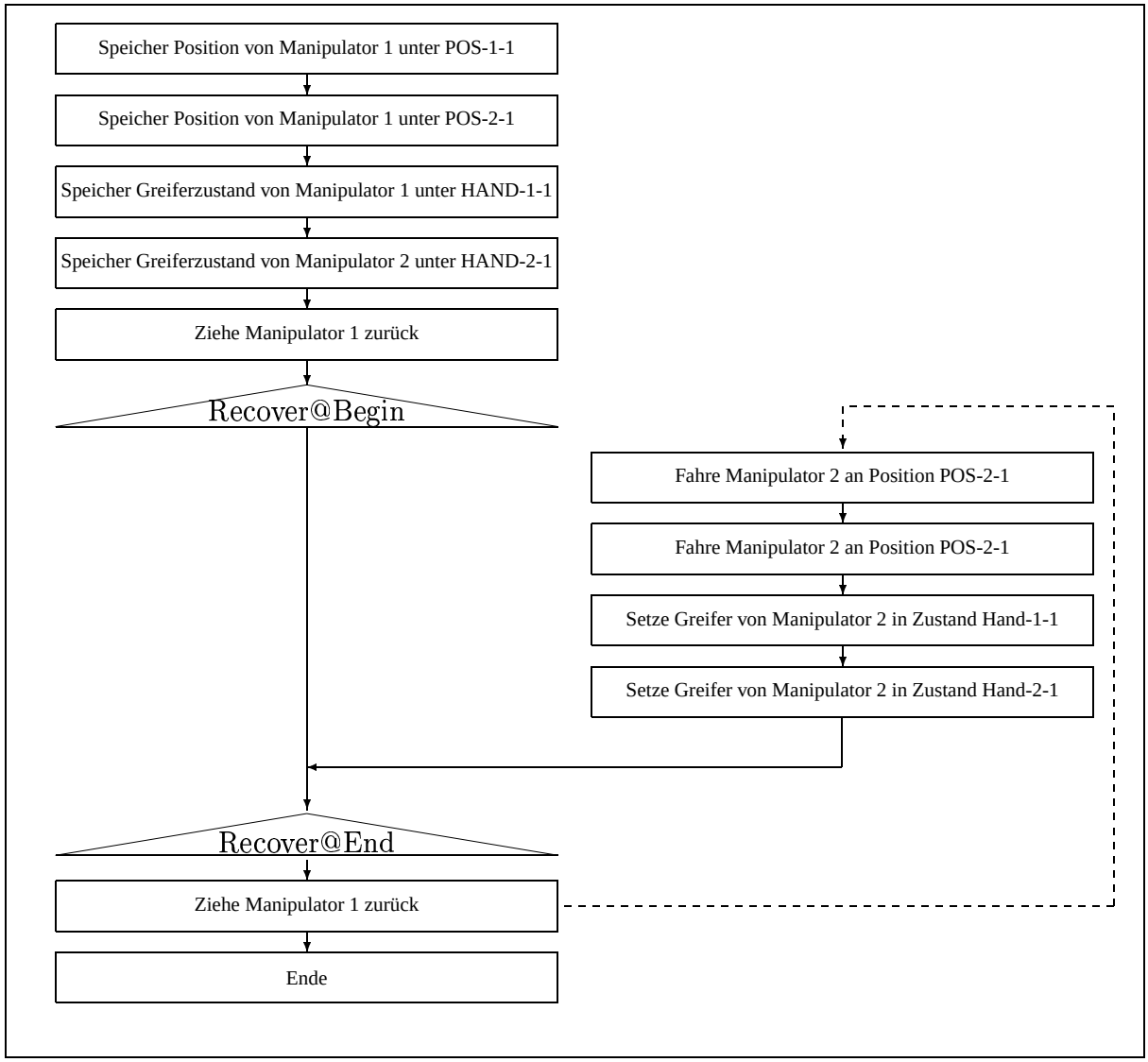


Abbildung 5.8: *Flussdiagramm einer Beispielsequenz zum Speichern der Position und des Greiferzustandes inkl.* Recover-*Abschnitt. Die Instruktionen zur Aktualisierung der internen Repräsentation sind zur besseren Übersicht in den Flussdiagrammen nicht dargestellt.*

Kapitel 6

Generalisierung

6.1 Arten der Generalisierung

In dieser Arbeit werden zwei Arten der Generalisierung behandelt.

1. Zusammenführen von gelernten unterschiedlichen Montagesequenzen für ein Aggregat bzw. Baugruppe.
2. Anwendung von bestehenden gelernten Montagesequenzen mit Variierung der verwendeten Bauteile.

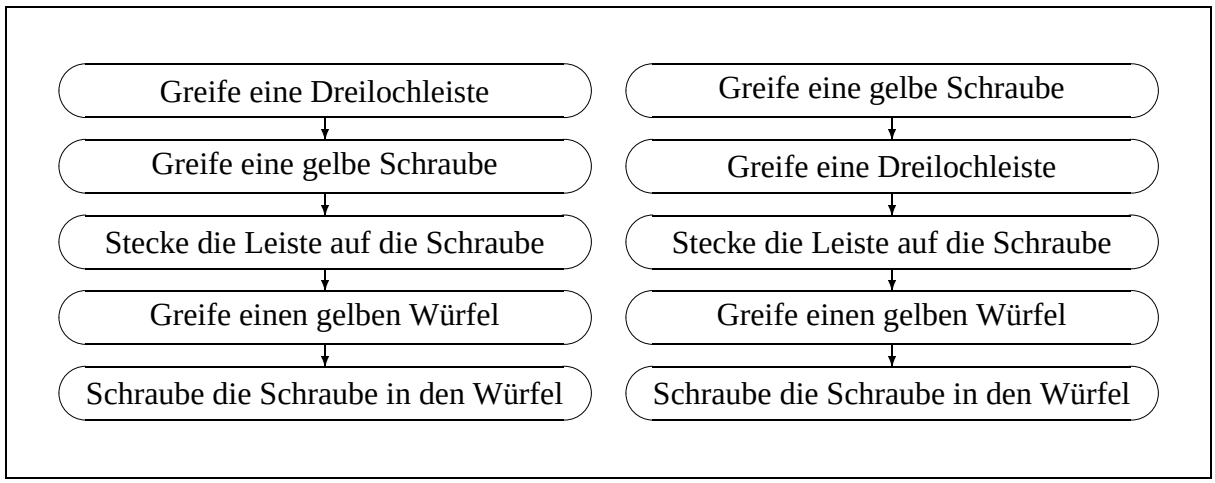


Abbildung 6.1: *Flussdiagramm zweier Sequenzen zum Bau eines Leitwerks.*

Wie mehrfach erwähnt, wird der Instrukteur im Regelfall mehrere Beispiele für den Zusammenbau eines Aggregates geben. Dabei ist nicht zu erwarten, dass die resultierenden Montagesequenzen identisch ausfallen. Zwei Ursachen kommen in Betracht; die Unterschiede ergeben sich durch unterschiedliche Szenarien und Vorkommnisse

während der Montage oder durch das Vorhandensein verschiedener wählbarer Wege zum Montageziel. Abb. 6.1 zeigt zwei unterschiedliche Sequenzen, die beide den Bau desselben Aggregates beschreiben.

Da a priori nicht bekannt ist, welche der vorhandenen Montagesequenzen bei einer späteren Wiederholung gewählt werden muss, sind die vorhanden Sequenzen zu fusionieren und so zu ändern, dass das Montagesystem selbstständig feststellen kann, wann welche Variante der Sequenz ausgeführt werden muss.

Ist die Montage eines Aggregates vom System erlernt worden, so sollen diese Sequenzen auch mit leicht unterschiedlichen Bauteilen zurechtkommen. Ist z.B. der Bau eines Leitwerks mit einer gelben Schraube, einer Dreilochleiste und einem Schraubwürfel erlernt worden, so ist es nur ein kleiner Unterschied, ob dies nun mit einer blauen Schraube gebaut wird. Es muss eine blaue statt einer gelben Schraube gegriffen werden.

6.2 Zusammenfassen von Sequenzen

6.2.1 Vergleichsmethode

Definition 6.2.1 *Zwei Operationsparameter des SI $x, y \in P$ sind gleich wenn gilt:*

$$x_i = y_i \quad \textit{für} \quad i = 1, \ldots, m \quad \Longleftrightarrow \quad x = y$$

Die Parameter x und y sind also gleich, wenn auch ihre jeweiligen Komponenten gleich sind.

Definition 6.2.2 *Zwei Instruktionen der SI $u, v \in C_M$ sind identisch, wenn gilt:*

$$M_u = M_v \quad \wedge \quad p_u = p_v \quad \Longleftrightarrow \quad u = v$$

Zwei Kommandos sind gleich wenn sowohl die Zustandstransformationsfunktionen M_u, M_v als auch die Parameter p_u, p_v identisch sind. Analog zu Definition 6.2.2 sind zwei Sequenzaufrufe identisch, wenn ihre Parameter identisch sind.

Definition 6.2.3

$$P_s \stackrel{\textit{def}}{=} \{(t, x_1, \ldots, x_n) \mid t \in T; x_i \in \Sigma\} \quad \textit{siehe Gl. 3.6}$$

Sei $u, v \in P_s$, dann ist $u = v$, wenn gilt

$$t_u = t_v \quad \wedge \quad x_{i,u} = x_{i,v}, \quad \textit{für} \quad i = 1, \ldots, n \quad \Longleftrightarrow \quad u = v$$

Zwei *IF*-Instruktionen (Operationen aus der Menge C_2 des SI) werden **immer** als Unterschiedlich betrachtet.

Definition 6.2.4 *Zwei Sequenzen π und ρ sind identisch, wenn gilt:*

$$\pi(z) = \rho(z) \quad \forall\, z \quad \Longleftrightarrow \quad \pi = \rho$$

Da eine durch Instruktionsanweisungen erlernte Sequenz nur Operationen aus den Mengen C_M und C_4 (Modulaufruf und Skriptaufruf) enthält, muss auf die restlichen Operationen der SI vorerst nicht eingegangen werden.

Der erste Schritt bei der Fusion zweier Sequenzen besteht in der Ermittlung der Positionen, an denen die Sequenzen identisch sind und an denen sie sich unterscheiden. Wie oben definiert, sind für das Montagesystem zwei Sequenzanweisungen gleich, wenn sowohl die Art der Operation[1] als auch die Parameter der Operation identisch sind. Diese Definition hat zur Konsequenz, dass Kommandos an den Roboter, die in der Realität keine unterschiedlichen Auswirkungen haben, vom Fusionsverfahren als unterschiedlichen angesehen werden. So ist z.B eine Bewegung entlang des Annäherungsvektors von $100mm$ oder $101mm$ in der Praxis nicht als unterschiedlich zu beurteilen. Der Grund liegt in der allgemeinen und flexiblen Grundstruktur von *OPERA*. Das System ist in weiten Teilen erweiterbar und so fehlt das semantische Wissen über die einzelnen Kommandos. Diese werden aber benötigt, um solche nur theoretisch bestehenden Unterschiede zu ignorieren.

Aus demselben Grund können auch Befehlsfolgen, die dieselben Auswirkungen haben, aber in unterschiedlicher Reihenfolge auftreten, nicht als gleich erkannt werden (siehe Beispiel Abb. 6.2). Das explizite Öffnen des Greifers ist überflüssig, da dies beim Greifen der 3-Lochleiste implizit geschieht. Es fehlt dem Montagesystem

- an Wissen über die Bedeutung der Parameter. Dies ist z.B. notwendig für die Beurteilung, welche Operation von welchem Manipulator ausgeführt wird, und
- an Wissen über die Auswirkungen einer Operation in der Realität. Die Auswirkungen von Instruktionen können abhängig oder unabhängig von ihrer Reihenfolge sein.

Allgemeine Vorgehensweise

Sind mehr als zwei Sequenzen zusammenzufassen, wird als erster Schritt die zweite Sequenz mit der ersten fusioniert. Hierfür werden die Instruktionen Position für Position miteinander verglichen. Als nächster Schritt wird die dritte Sequenz mit der nun modifizierten ersten zusammengefasst u.s.w. . Die hinzuzufügende Sequenz wird nachfolgend Quellsequenz ρ und die Sequenz in die alle Sequenzen eingefügt werden, Zielsequenz π genannt. Sind keine Unterschiede zwischen Ziel-und Quellsequenzen vorhanden, so werden nur die Sensordaten der jeweiligen Instruktionen in die Zielsequenz übertragen. Eine Zielsequenzinstruktion besitzt anschließend nicht nur ein Beispiel für den Sensorzustand des Systems sondern mehrere.

[1] Aufruf einer Sequenz oder eines bestimmten Moduls.

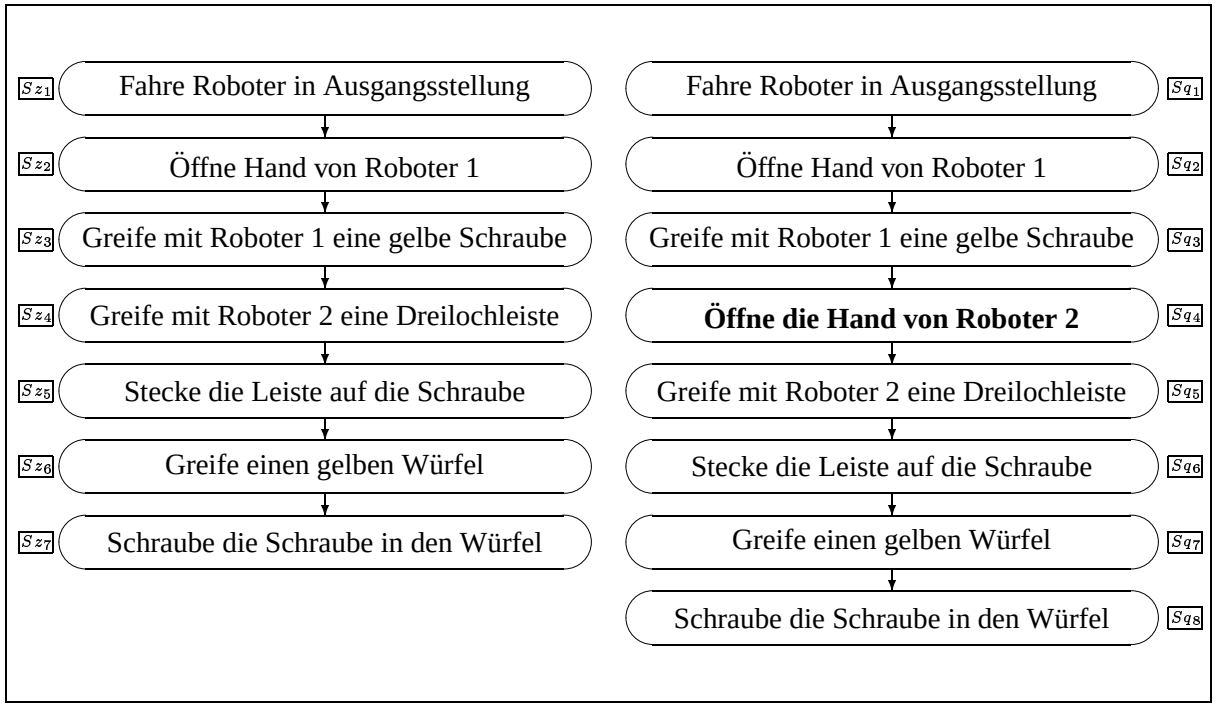


Abbildung 6.2: *Beispiel für zusätzliche Instruktionen. Sz_i und Sq_i repräsentieren die aufgezeichneten Sensorzustände der i-ten Instruktion.*

6.2.2 Erzeugen von Verzweigungen

Unterscheiden sich zwei oder mehrere Sequenzen obwohl das Endprodukt ein- und dasselbe Aggregat ist, so gibt es zwei Gründe hierfür:

1. Die Reihenfolge der Operationen ist teilweise beliebig und der Instrukteur hat davon Gebrauch gemacht (z.B. siehe Abb. 6.1).

2. In der realen Szene ist eine andere Situation eingetreten als bei den anderen Beispielen und daher musste die Montage mit unterschiedlichen Operationen fortgesetzt werden. Dies sind z.B. Ausrichtoperationen.

In beiden Fällen müssen Verzweigungen[2] generiert und mit dem zusätzlichen Handlungsstrang in die Sequenz eingefügt werden.

Ist eine Sequenzposition z ermittelt worden, an der Ziel- und Quellsequenz unterschiedlich sind

$$\pi(z) \neq \rho(z), \tag{6.1}$$

so wird im nächsten Schritt herausgefunden, wo die beiden zu vergleichenden Sequenzen wieder identisch sind. Im folgenden werden nur die Teilsequenzen betrachtet, auf

[2]Operationen aus der Menge C_2 des SI.

denen sich die Quell- und die Zielsequenzen unterscheiden. Es sind folgende Fälle zu betrachten:

1. Die Quellsequenz besitzt zusätzliche Instruktionen gegen über der Zielsequenz (siehe Abb 6.2).
2. Die Zielsequenz besitzt gegenüber der Quellsequenz zusätzliche Instruktionen.
3. Beide Sequenzen unterscheiden sich in einer Teilsequenz (siehe Abb. 6.3).

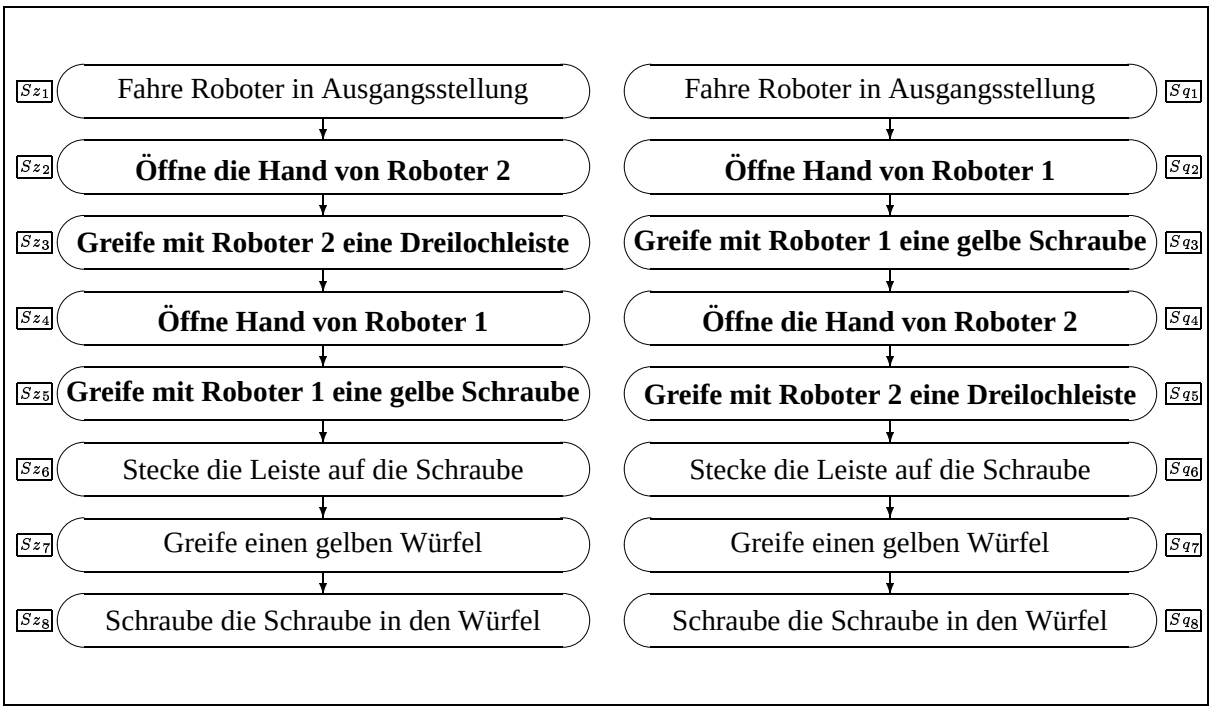


Abbildung 6.3: *Beispiel zweier unterschiedlicher Teilsequenzen (fett).* Sz_i *und* Sq_i *repräsentieren die aufgezeichneten Sensorzustände der* i*-ten Instruktion.*

Andere Kombinationen können nicht auftreten. Bevor eine Verzweigung in die Zielsequenz eingebaut werden kann, muss erst die Positionen ermittelt werden, ab der die Quell- und Zielsequenz wieder identisch sind. Daher wird erst geprüft, ob folgende Aussage gilt:

$$\exists\, i > z \quad \text{für das gilt} \quad \pi(z) = \rho(i) \tag{6.2}$$

Es ist allerdings nicht ausreichend zu prüfen, ob nur eine einzelne Instruktion identisch ist, da

1. beim Lernvorgang für eine Anweisung des Benutzers immer mindestens zwei Instruktionen der SI verwendet werden (siehe Kapitel 5.1.2);

2. in unterschiedlichen Montagevarianten oft dieselben Instruktionen verwendet werden, ohne dass die nachfolgenden Instruktionen identisch sind.

Erst wenn mindestens 4 aufeinanderfolgende Instruktionen identisch sind, wird davon ausgegangen, dass die nachfolgenden Instruktionen zu einer größeren identischen Teilsequenz gehören[3]. Sei

$$b = 4, 6, 8, \ldots$$

Die Bedingung (6.2) ändert sich daher wie folgt:

$$\exists\, j > z \quad \text{für das gilt} \quad \pi(z+m) = \rho(j+m) \quad \text{mit} \quad m = 0, \ldots, b \tag{6.3}$$

Abb. 6.4 illustriert zwei unterschiedliche Sequenzen bei der die Aussage (6.3) gilt. Die

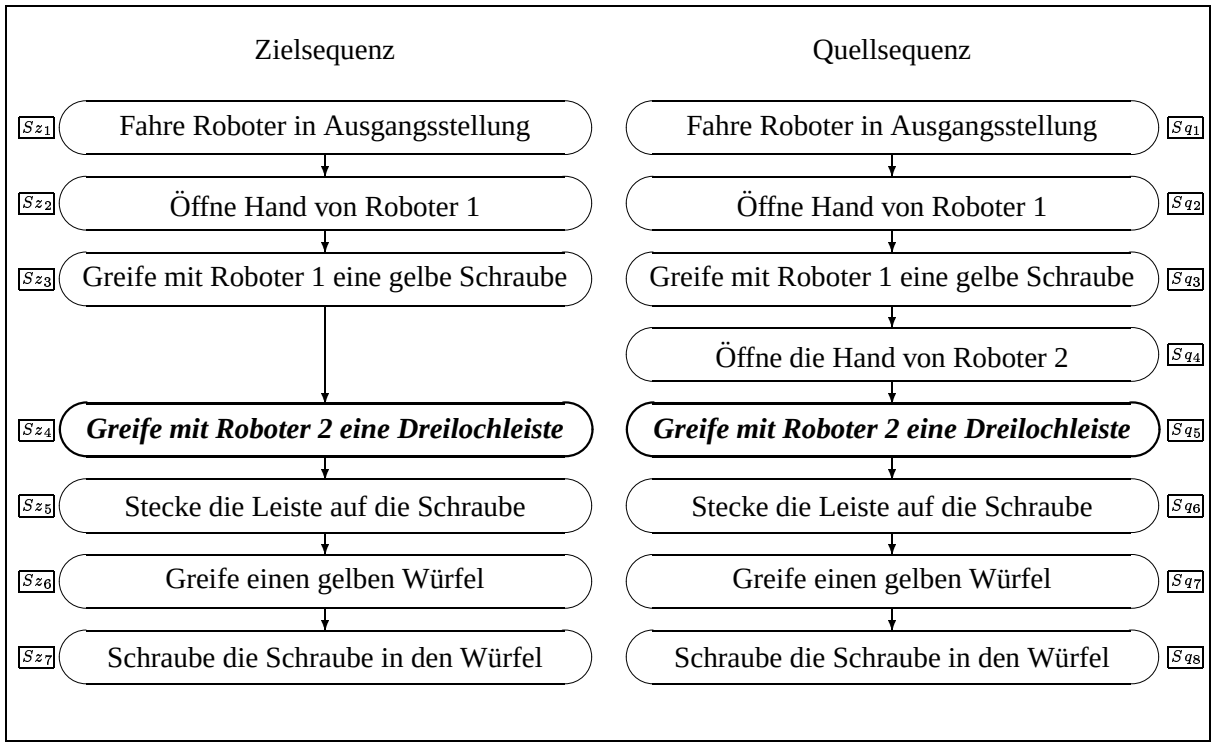


Abbildung 6.4: *Erfüllung der Bedingung (6.3).* Sz_i *und* Sq_i *repräsentieren die aufgezeichneten Sensorzustände der* i*-ten Instruktion.*

fett markierten Knoten zeigen die entsprechenden Instruktionen $\pi(z)$ und $\rho(j)$. Existiert kein i, das diese Bedingung (6.3) erfüllt, so wird geprüft, ob die Zielsequenz gegenüber der Quellsequenz mehr Operationen enthält.

$$\exists\, i > z \quad \text{für das gilt} \quad \pi(i+m) = \rho(z+m) \quad \text{mit} \quad m = 0, \ldots, b \tag{6.4}$$

[3]In den gezeigten Flussdiagrammen sind die Instruktionen zum Abgleich der internen Repräsentation (*Update Scene*) zur besseren Übersicht nicht dargestellt.

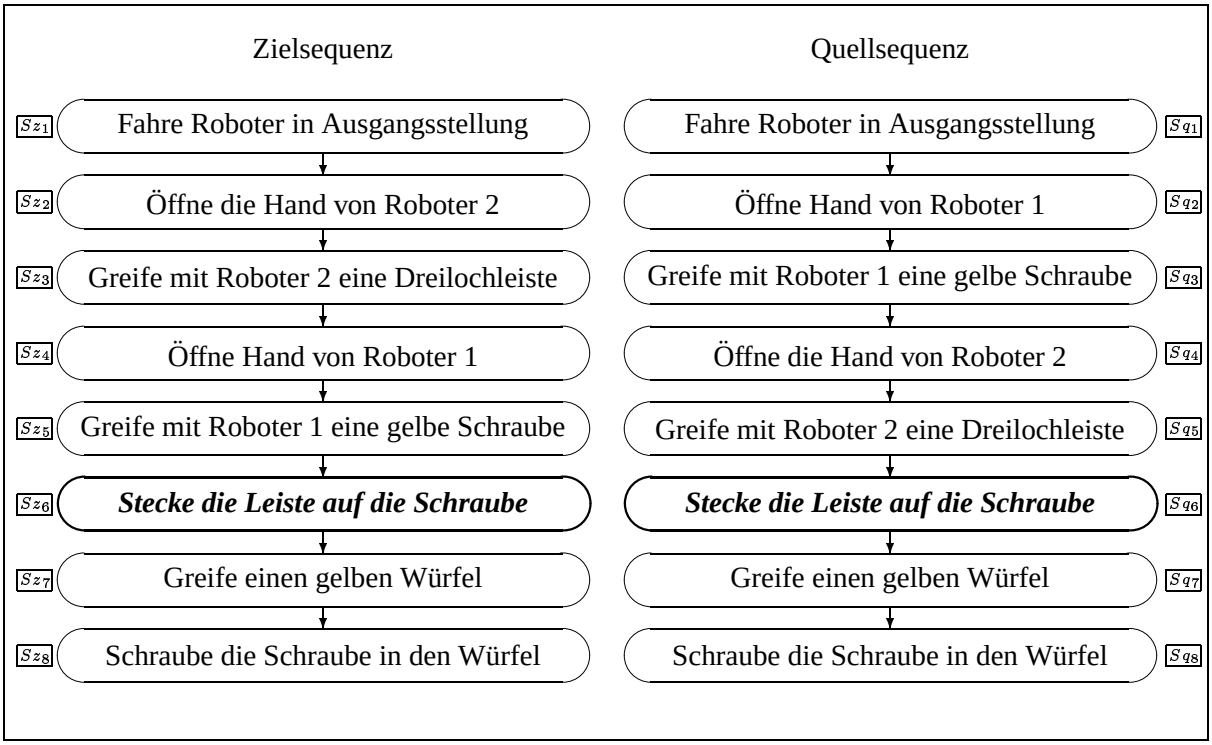


Abbildung 6.5: *Erfüllung der Bedingung (6.5). Sz_i und Sq_i repräsentieren die aufgezeichneten Sensorzustände der i-ten Instruktion.*

Ist auch die Bedingung (6.4) nicht erfüllt, dann existieren zwei unterschiedliche Teilsequenzen in der Ziel- und Quellsequenz, und es wird daher geprüft, ob folgende Bedingung erfüllt ist:

$$\exists\, i, j > z \quad \text{für das gilt} \quad \pi(i+m) = \rho(j+m) \quad \text{mit} \quad m = 0, \ldots, b \tag{6.5}$$

Abb. 6.5 illustriert zwei unterschiedliche Sequenzen bei der die Aussage (6.5) gilt. Die fett markierten Knoten zeigen die entsprechenden Instruktionen $\pi(i)$ und $\rho(j)$. Ist auch diese Bedingung (6.5) nicht erfüllt, so sind beide Restsequenzen unterschiedlich

$$\pi(z+i) \neq \rho(z+i) \quad \text{für } i \geq z$$

und beiden Handlungsstränge können erst am Ende zusammengeführt werden. Allgemein kann die Bedingung wie folgt definiert werden[4]:

$$\exists\, i, j \geq z \quad \text{für die gilt} \quad \pi(i+m) = \rho(j+m) \quad \text{mit} \quad m = 0, \ldots, b \tag{6.6}$$

Existieren mehrere unterschiedliche Teilsequenzen, so kann nach dem Einfügen der ersten Verzweigung der Vergleich von die Quell- und Zielsequenz nicht mehr mit demselben Index z stattfinden. Daher wird zwischen z_Q für die Quellsequenz und z_Z für die

[4]Es ist zu beachten, dass hier mehrere mögliche algorithmische Umsetzungen möglich sind, die u. U. unterschiedliche Resultate liefern.

Zielsequenz unterschieden. Die Aussage (6.1) ändert damit zu

$$\pi(z_Z) \neq \rho(z_Q),$$

und die allg. Suchbedingung (6.6) zu:

$$\exists\, i \geq z_Z \;\wedge\; j \geq z_Q \quad \text{für das gilt} \quad \pi(i+m) = \rho(j+m) \quad \text{mit} \quad m = 0, \ldots, b \qquad (6.7)$$

η bezeichnet die neu entstehende Sequenz, welche vor dem Einfügen der Instruktionen aus ρ mit π identisch ist

$$\forall i \quad \pi(i) = \eta(i)$$

und nach dem Einfügen zur neuen Zielsequenz $\eta \longrightarrow \pi$ wird. Nach dem Finden der Position von dem an die Ziel- und Quellsequenz wieder identisch sind, kann die Verzweigung in die Zielsequenz eingebaut werden. Hierzu wird zuerst an der Position z_Z der Zielsequenz eine *IF*-Instruktion eingefügt, deren boolsche Funktion im Bedingungsteil vorerst undefiniert bleibt. Damit ist $\eta(z_Z + 1) = \pi(z_Z)$.

Fall 1

Besitzt die Quellsequenz ρ gegenüber der Zielsequenz π zusätzlich Instruktionen, so müssen diese in die Zielsequenz ab der Position z_Z eingebaut werden. Nach dem Einbau ist die neue Zielsequenz η wie folgt definiert:

$$\begin{array}{lcl}
\eta(k) & = & \pi(k) \quad \text{für} \quad k = z_Z - 4, \ldots, z_Z - 1 \\
\eta(z_Z) & = & \tau_{\mathsf{IF}} \\
\eta(z_Z + 1) & = & \rho(z_Q) \\
\eta(z_Z + 2) & = & \rho(z_Q + 1) \\
 & \vdots & \\
\eta(z_Z + (j - z_Q) + 1) & = & \rho(j - 1) \\
\eta(z_Z + (j - z_Q) + 2) & = & \pi(z_Z) \\
\eta(z_Z + (j - z_Q) + 3) & = & \pi(z_Z + 1) \\
 & \vdots &
\end{array}$$

τ_{IF} ist die eingefügte *IF*-Instruktion

$$\tau_{\mathsf{IF}} \in C_2\ .$$

Abb. 6.6 zeigt den neu entstandene Sequenzabschnitt als Flussdiagramm und Abb. 6.7 die resultierende Sequenz für das Beispiel aus Abb. 6.4.

Fall 2

Für den Fall, dass die Quellsequenz weniger Instruktionen als die Zielsequenz besitzt, muss nur eine *IF*-Instruktion an Position z_Z eingefügt werden.

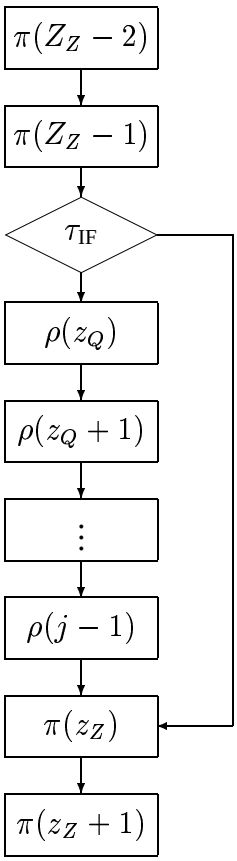


Abbildung 6.6: Teilabschnitt der neuen Zielsequenz η als Flussdiagramm (Fall 1).

Fall 3

Für den Fall, dass Ziel- und Quellsequenz unterschiedliche Handlungsstränge und damit unterschiedliche Teilsequenzen besitzen, muss ein *IF-THEN-ELSE* Konstrukt eingebaut werden. Als erster Schritt wird an Position z_Z die *IF*-Instruktion eingefügt; anschließend die Instruktionen $\rho(z_Q), \ldots, \rho(j)$ und eine Sprungoperation. Die resultierende Teilsequenz der neuen Sequenz η setzt sich dann wie folgt zusammen:

$$
\begin{array}{lcl}
\eta(k) & = & \pi(k) \quad \text{für} \quad k = z_Z - 4, \ldots, z_Z - 1 \\
\eta(z_Z) & = & \tau_{\text{IF}} \\
\eta(z_Z + 1) & = & \rho(z_Q) \\
 & \vdots & \\
\eta(z_Z + j - z_Q) & = & \rho(j - 1) \\
\eta(z_Z + j - z_Q + 1) & = & \tau_{\text{Jump}} \\
\eta(z_Z + j - z_Q + 2) & = & \pi(z_Z) \\
 & \vdots &
\end{array}
$$

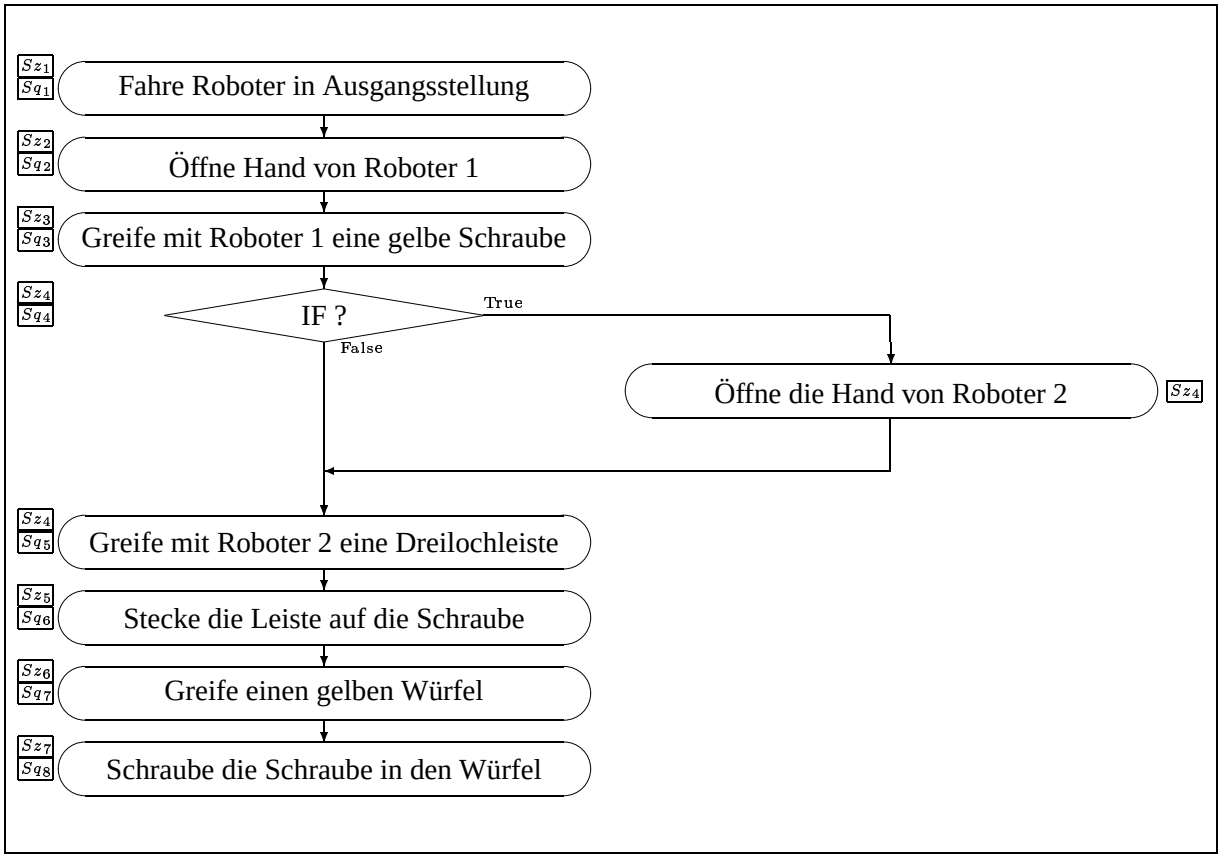


Abbildung 6.7: *Flussdiagramm der resultierende Sequenz für Fall 1.* Sz_i *und* Sq_i *repräsentieren die abgelegten Sensorzustände der* i*-ten Instruktion. Es ist zu erkennen, wie die aufgezeichneten Sensordaten nach der Zusammenlegung von zwei Sequenzen zugeordnet werden. Der* IF*-Instruktion werden die Sensordaten der möglichen Nachfolgeinstruktion zugeordnet, da beide Zustände vor dieser Verzweigung aufgetreten sind.*

mit $\tau_{\mathsf{IF}} \in C_2$, $\tau_{\mathsf{Jump}} \in C_1$. Abb. 6.8 zeigt den neu entstandenen Sequenzabschnitt als Flussdiagramm und Abb. 6.9 die resultierende Sequenz für das Beispiel aus Abb.6.5.

Sind die Verzweigungen eingebaut, so müssen abschließend die Sensordaten von $\rho(z_Q)$ als auch von $\pi(z_Z)$ der bei z_Z eingefügten *IF*-Instruktion hinzugefügt werden. Sowohl die Sensordaten von $\pi(z_Z) = \eta(z_Z + (j - z_Q) + 2)$ als auch die von $\rho(z_Q)$ beschreiben vor der Verzweigung mögliche Sensorzustände.

Berücksichtigung von bestehenden *IF*-Instruktionen

Ein Sonderfall ist gegeben, wenn Teilsequenzen der Ziel- und die Quellsequenz nur deshalb nicht identisch sind, weil in der Zielsequenz schon eine Verzweigung eingebaut worden ist. Da der Instrukteur keine Anweisungen geben kann, die Bedingungen enthalten, können solche Instruktionen in der Quellsequenz nicht auftreten. Abb. 6.10 zeigt

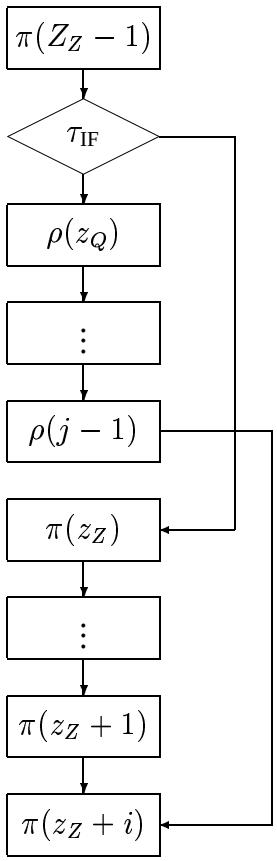


Abbildung 6.8: Teilabschnitt der neue Zielsequenz η als Flussdiagramm (Fall 3).

eine solche Situation. Nur die *IF*-Instruktion selbst und die beiden Instruktionen des THEN-Teils (fett markiert) unterscheiden sich von der Quellsequenz. Wird dies nicht berücksichtigt, würde für jede weitere Quellsequenz eine neue und überflüssige Verzweigung eingefügt werden. Tritt die Situation auf, dass

$$\pi(z_Z) \neq \rho(z_Q) \quad \text{und} \quad \pi(z_Z) \in C_2,$$

dann muss überprüft werden, ob die Teilsequenz

$$\rho(i) \quad \text{für} \quad i = z_Q, \ldots, j$$

identisch mit der Teilsequenz des *THEN*-Abschnittes oder soweit existent identisch mit dem *ELSE*-Abschnitt der Zielsequenz ist. Ist dies der Fall, so werden nur die beim Lernprozess gespeicherten Sensordaten in der Quellsequenz in die entsprechenden Instruktionen der Zielsequenz übernommen und die Zielsequenz wird ansonsten unverändert gelassen. Ist die Teilsequenz nicht mit dem *IF*- oder *THEN*-Abschnitt der Zielsequenz identisch so wird wie oben beschrieben eine Verzweigung aufgebaut.

Andere Fälle wie die hier beschriebenen können nicht auftreten, so dass dieses Fusionsverfahren in jedem Fall alle Beispielsequenzen korrekt zusammenfasst.

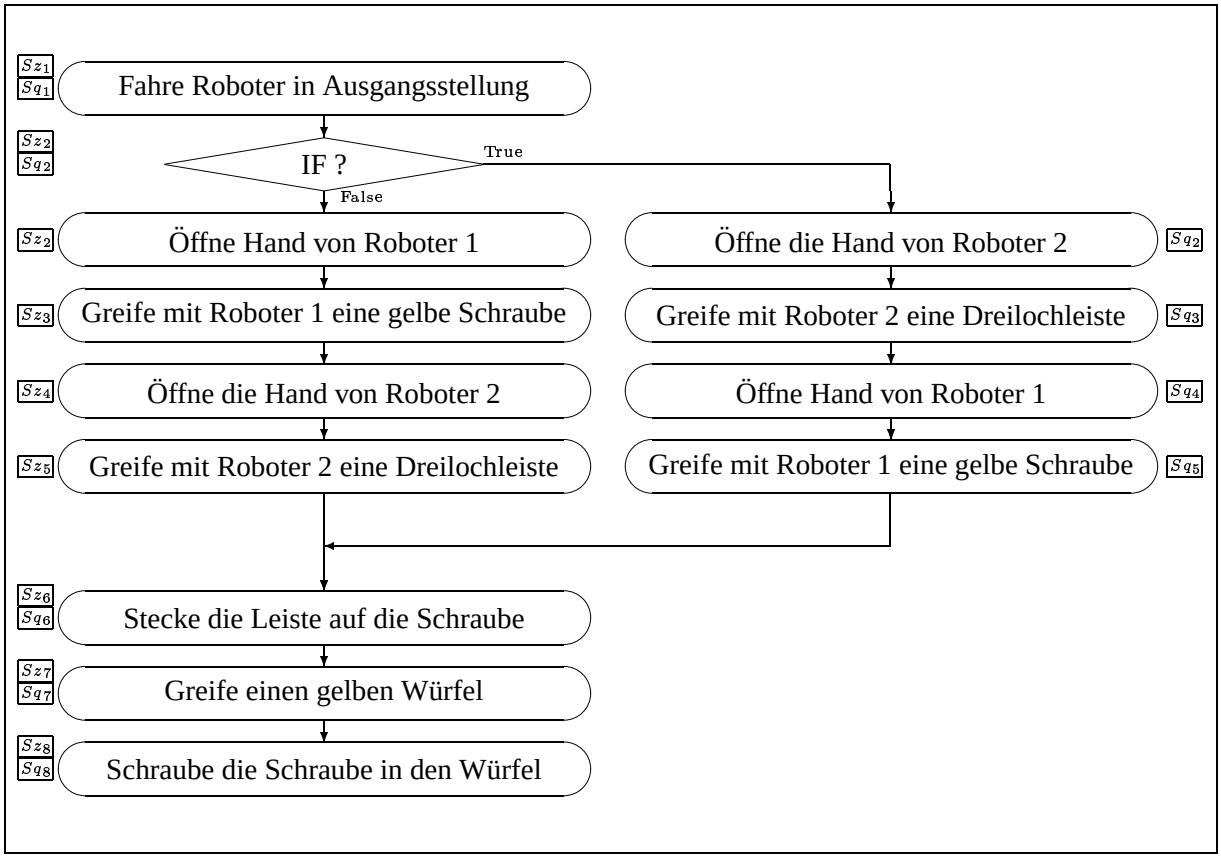


Abbildung 6.9: *Flussdiagramm der resultierende Sequenz für Fall 3. Sz_i und Sq_i repräsentieren die abgelegten Sensorzustände der i-ten Instruktion. Es ist zu erkennen, wie die aufgezeichneten Sensordaten nach der Zusammenlegung von zwei Sequenzen zugeordnet werden. Der* IF-*Instruktion werden die Sensordaten der möglichen Nachfolgeinstruktion zugeordnet, da beide Zustände vor dieser Verzweigung aufgetreten sind.*

6.2.3 Auswertung von Sensormustern

Sind alle verfügbaren Sequenzen für die Montage eines Aggregates zusammengefasst, müssen die eingefügten *IF*-Instruktionen vervollständigt werden. Die boolsche Funktion dieser Instruktionen ist noch undefiniert. Die jeweilige Entscheidung, ob bei einer Verzweigung der eine oder der andere Handlungsstrang genommen wird, kann das Montagesystem nur anhand von Sensordaten ausfindig machen. Es muss

- ermittelt werden, welche der zur Verfügung stehenden Sensoren für eine Entscheidung zu berücksichtigen sind, und
- ein Regler gebaut werden, der die Daten der berücksichtigten Sensoren fusioniert und eine Entscheidung fällt.

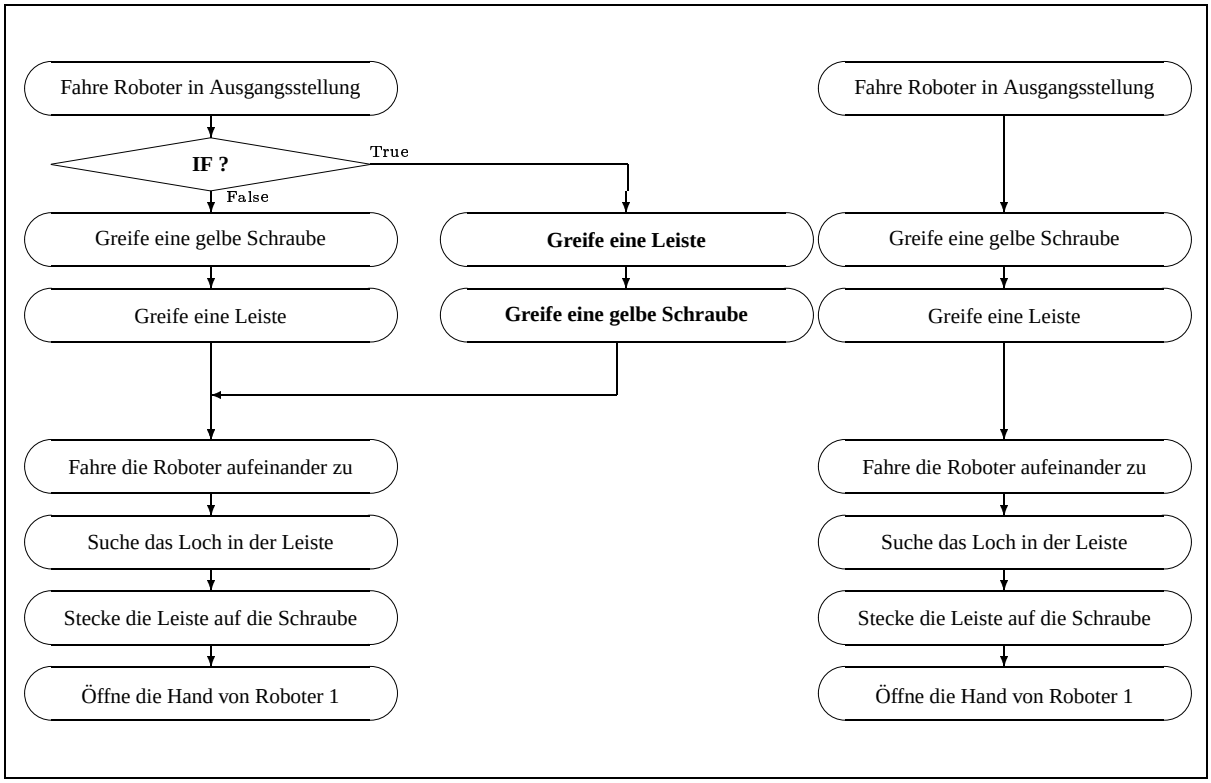


Abbildung 6.10: *Beispiel für Differenzen aufgrund von Verzweigungen.*

Auswahl der Sensoren

Für die Auswahl der benötigten Sensoren werden die zu den jeweiligen Instruktionen gespeicherten Sensorzustände betrachtet. Durch das Zusammenführen mehrerer Sequenzen können einer Instruktion auch mehrere Sensorzustände zugeordnet sein. Interessant sind die Zustände der jeweilig ersten Instruktion des *THEN* und des *ELSE*-Teils einer Verzweigung (in Abb.6.11 fett unterlegt).

Diesen Instruktionen sind die jeweiligen Sensorzustände für den einen oder anderen Handlungsstrang zugeordnet. Sollte sich eine Entscheidung, welche Teilsequenz auszuführen ist, anhand der verfügbaren Sensordaten ermitteln lassen, so müssen sich die Zustände oder Teile unterscheiden. Der Sensorzustand S_Z sei definiert als

$$S_Z \stackrel{\text{def}}{=} \{(r_i, t_i)^n \mid r_i \in \mathbb{G}^m; t_i \in T; m, n \in \mathbb{Z}\}$$

t_i bezeichnet den Namen eines Sensors und r_i den dazugehörigen Wert. So bezeichnet z.B der Name “RobotPos-x-1” die x-Position des ersten Roboter. Die könnte den Wert $100,05$ haben. Der nächste Verarbeitungsschritt besteht darin, die relevanten Elemente der jeweiligen Sensorzustände $s_{\text{THEN}}, s_{\text{ELSE}} \subset S_Z$ zu ermitteln. Da davon ausgegangen werden muss, dass die Dimension der Elemente von s_{THEN} und s_{ELSE} ungleich sind,

$$\dim a \neq \dim b \quad \text{mit} \quad a \in s_{\text{THEN}} \ \wedge \ b \in s_{\text{ELSE}},$$

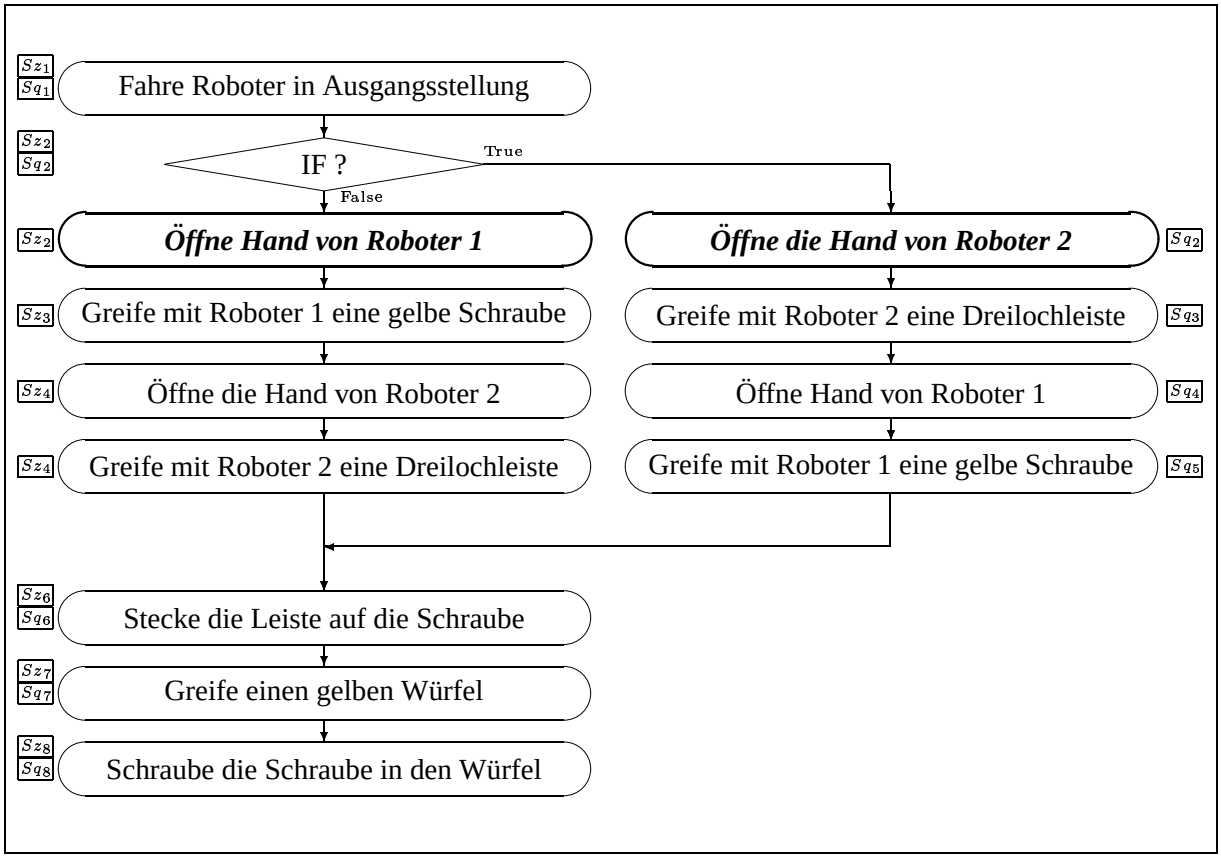


Abbildung 6.11: *Für die Berechnung der boolschen Funktion betrachtete Instruktionen (fett).* Sz_i *und* Sq_i *repräsentieren die abgelegten Sensorzustände der* i*-ten Instruktion. Dir Sensordaten der Zustände* Sz_2 *und* Sq_2 *werden für die Berechnung der boolschen Funktion herangezogen.*

muss erst eine Dimensionsangleichung durchgeführt werden. Als erstes können alle Sensordaten unberücksichtigt bleiben, die nicht in allen gespeicherten Zuständen vorkommen. Nach diesem Verarbeitungsschritt existieren s'_{THEN} und s'_{ELSE} mit

$$\dim a' = \dim b' \quad \text{mit} \quad a' \in s_{\text{THEN}} \ \wedge \ b' \in s_{\text{ELSE}}$$

Beispiel

Tabelle 6.1 zeigt den Vorgang exemplarisch anhand von fünf Beispielzuständen. Die Sensorwerte in den Namen *Position X, Position Y, Position Z* können für eine weitere Auswertung in Betracht gezogen werden. Der Sensorwert mit dem Namen *Kraft in N* ist dagegen nicht in allen Zuständen vorhanden und fällt daher heraus. Selbst wenn er Informationen beinhalten würde, kann er nicht zur Entscheidung herangezogen werden, da nicht sicher ist, dass die Daten von diesem Sensor immer zur Verfügung stehen.

Sensorname	Zustand 1	Zustand 2	Zustand 3	Zustand 4	Zustand 5	berücksichtigte Sensoren
Position X	$100,0$	$100,05$	$99,65$	$99,0$	$89,0$	X
Position Y	$300,1$	$301,05$	$299,45$	$-299,0$	$-289,0$	X
Position Z	$10,0$	$10,0$	$10,0$	$10,0$	$10,0$	X
Kraft in Z	$0,0$	$1,01$	–	$9,0$	$8,0$	

Tabelle 6.1: Beispiel zur Auswertung von Sensoren. Nur diejenigen Sensoren werden für die Auswertung betrachtet, die auch in allen Beispielzuständen vorkommen. Bei *Kraft in Z* ist dies nicht der Fall.

Im nächsten Schritt kann die Dimension der Elemente von s'_{THEN} und s'_{ELSE} weiter reduziert werden. So müssen Sensordaten, deren Werte keine Varianz aufweisen, ebenfalls nicht berücksichtigt werden.

Beispiel

Tabelle 6.2 zeigt den Vorgang anhand der oben gezeigten Beispielzustände. Der Sensor *Kraft in Z* ist nicht mehr vorhanden, da er im ersten Schritt gestrichen worden ist. Es

Sensorname	Zustand 1	Zustand 2	Zustand 3	Zustand 4	Zustand 5	berücksichtigte Sensoren
Position X	$100,0$	$100,05$	$99,65$	$99,0$	$89,0$	X
Position Y	$300,1$	$301,05$	$299,45$	$-299,0$	$-289,0$	X
Position Z	$10,0$	$10,0$	$10,0$	$10,0$	$10,0$	

Tabelle 6.2: Beispiel zur Auswertung von Sensoren. Nur diejenigen Sensoren werden für die Auswertung betrachtet, die eine Varianz aufweisen. Bei *Position Z* ist dies nicht der Fall.

bleiben nur noch die Variablen *Position X* und *Position Y* übrig, da der Sensor *Position Z* keine Varianz besitzt und damit keine Information über die Unterschiede der Zustände beisteuert. Anschließend wird die Korrelation zwischen den Daten einzelner Sensoren und dem zugeordneten Handlungspfad berechnet. Stellt sich dabei heraus, dass die jeweiligen Sensordaten mit dem zugeordneten Handlungspfad nicht oder nur schwach korreliert sind, so werden auch sie nicht berücksichtigt.

Die verbleibenden Sensordaten werden dann als Eingang für den in Kapitel 4.4 vorgestellten Regler verwendet. Im Gegensatz zu der im Kapitel 4.5 vorgestellten Anwendung, soll der Regler nur eine boolsche Entscheidung treffen.

Beispiel

Das Verfahren soll exemplarisch am Ausrichten einer Leiste gezeigt werden. Nach dem Zusammenbau eines Schraubwürfels, einer Leiste und einer Schraube muss die Leiste am Schraubwürfel senkrecht ausgerichtet werden. Je nach Winkel der Leiste muss der Roboter eine von zwei möglichen Trajektorien abfahren. Mögliche Szenen sind

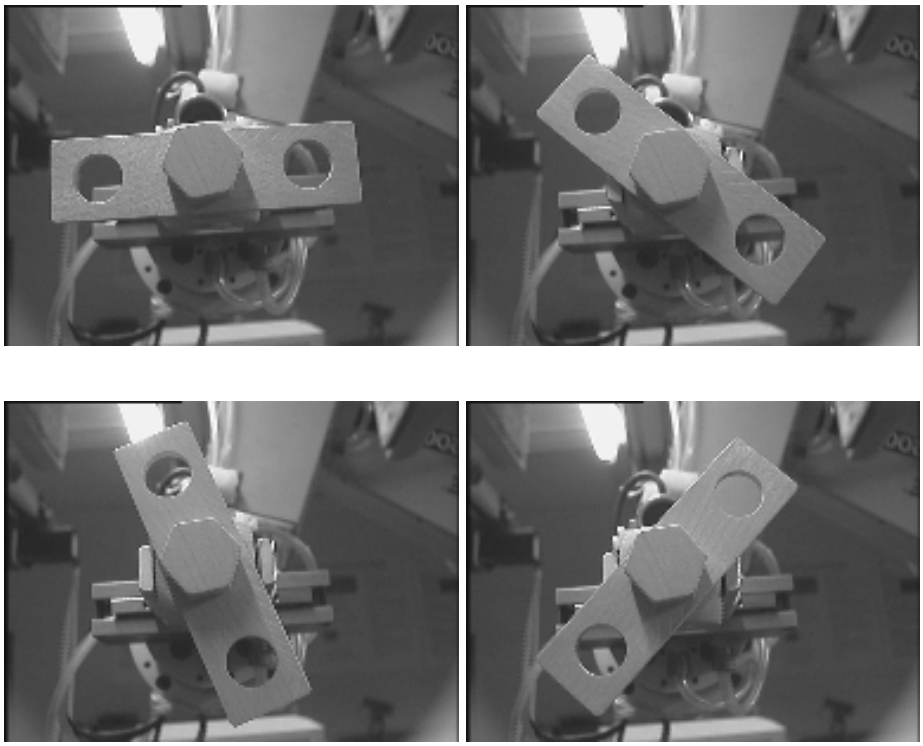


Abbildung 6.12: Beispielsituation vor dem Ausrichten einer Leiste.

in Abb. 6.12 dargestellt. Sie zeigen das noch nicht fertige Leitwerk (die Leiste ist noch ausgerichtet) aus der Sicht der Handkamera des anderen Roboters. Sind dem Montagesystem für die jeweiligen Situationen die entsprechenden Handlungsstränge durch den Instrukteur gezeigt worden, besteht an diesem Punkt der Montage eine Verzweigung bei der das System anhand seiner Sensordaten entscheiden muss, welche Bewegung zum Ausrichten der Leiste durchgeführt werden muss. In diesem Beispiel stehen dem Montagesystem folgende Informationen zur Verfügung (Ausschnitt siehe Tabelle 6.3):

- Bild der Handkamera vom Roboter, der ausrichten soll (Abb. 6.12),
- Kraftwerte der beiden Roboter und
- Boolscher Wert, ob der Manipulator etwas in der Hand hält.

Zunächst müssen die Sensoren, die die benötigten Informationen liefern, nach dem oben beschriebenen Verfahren herausgefunden werden. Die Berechnung der Korrelation ergibt die Werte die Tabelle 6.4 aus denen entnommen werden kann, dass nur die Handkamera Information liefert mit deren Hilfe die Situation bewertet werden kann. Alle anderen Sensordaten zeigen keine hohe Korrelation und brauchen daher nicht berücksichtigt zu werden. Abb. 6.13(b) zeigt den Projektionsvektor der aus den Bild der Handkameras mit ORF berechnet wurde. Dies führt zu einem Regler mit nur einem Eingang

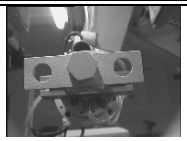
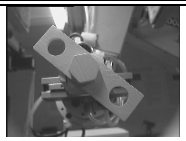
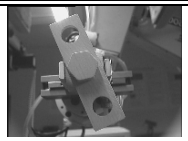
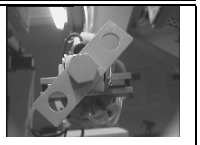
	Bsp.1	Bsp.2	···	Bsp.49	Bsp.50
Handkamera			···		
Kraft N Roboter 1	0.01	0.063	···	0.038	−0.01
Kraft O Roboter 1	−0.23	−0.24	···	−0.28	−0.23
Kraft A Roboter 1	0.11	0.12	···	−0.1	0.16
Kraft N Roboter 2	−0.16	−0.16	···	−0.16	−0.12
Kraft O Roboter 2	−0.05	−0.03	···	−0.03	−0.08
Kraft A Roboter 2	−0.10	−0.21	···	−0.32	−0.42
Handstatus Roboter 1	0	0	···	0	0
Handstatus Roboter 2	1	1	···	1	1
Sollwert	1	1	···	1	0

Tabelle 6.3: Sensorzustände der einzelnen Situationen bzw. Beispiele.

(siehe Abb. 6.14).

Zusammenfassung

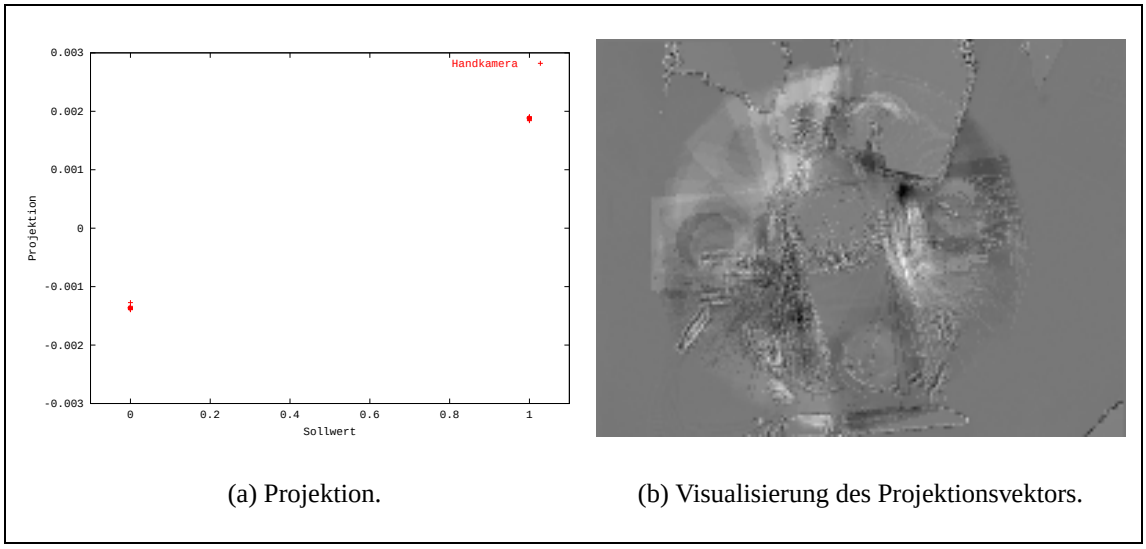
Die Vervollständigung der *IF*-Instruktionen in der fusionierten Sequenz (Zielsequenz) läuft in vier Schritten ab:

1. Bestimmen aller Sensordaten, die in allen Sequenzbeispielen vorkommen,

2. Bestimmen der Sensordaten, die eine Varianz aufweisen,

	Korrelation
Handkamera	0.99
Kraft N Roboter 1	0.69
Kraft O Roboter 1	0.08
Kraft A Roboter 1	0.00
Kraft N Roboter 2	0.14
Kraft O Roboter 2	0.16
Kraft A Roboter 2	0.06
Handstatus Roboter 1	0.00
Handstatus Roboter 2	0.00

Tabelle 6.4: Korrelation der einzelnen Sensordaten mit den Solldaten.



(a) Projektion. (b) Visualisierung des Projektionsvektors.

Abbildung 6.13: Reduktion der Sensordaten von der Handkamera.

3. Auswahl derjenigen Sensordaten, die eine signifikante Korrelation mit den Solldaten s_i ($s_i \in \{0, 1\}$) aufweisen, und

4. Erstellen eines Reglers nach Kapitel 4.4, durch den eine boolsche Funktion repräsentiert wird.

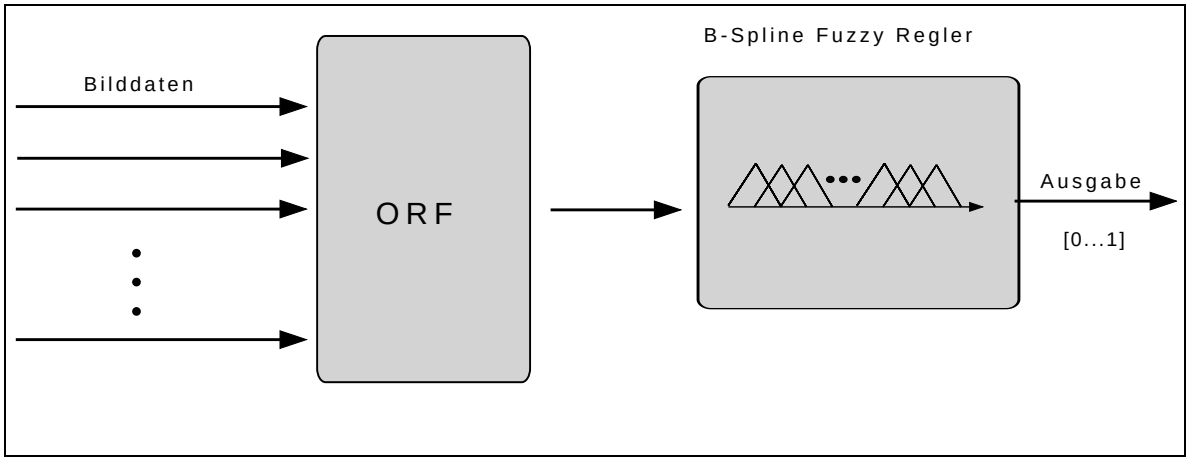


Abbildung 6.14: *Aufbau des Entscheidungcontrollers.*

Kann zwischen den Sollwerten und den Sensordaten keine Korrelation festgestellt werden, so kann eine Auswahl des Handlungsstranges nicht anhand von Sensordaten entschieden werden. Dies kann zwei Gründe haben: Es ist unerheblich, ob der eine oder der andere Handlungsstrang genommen wird, oder es liegen nicht die notwendigen Sensordaten vor. In diesem Fall wird ein Zufallsentscheid getroffen. Eine andere

Möglichkeit besteht darin, den Instrukteur die boolsche Funktion definieren zu lassen. Da das Montagesystem dem Instrukteur nur schwer vermitteln kann, welche Entscheidung gefällt werden muss, ist die Möglichkeit nur bei einfachen oder nur sehr wenigen Beispielsequenzen durchführbar.

6.2.4 Retrieval

Ist die Generalisierung der Beispielsequenzen für ein Aggregat abgeschlossen, wird die Sequenz abgelegt und kann zu einem späteren Zeitpunkt wiederholt werden. Zusätzlich zu der Handlungssequenz wird der Anfangszustand des Montagesystems mit abgelegt. Bei einem Abruf der Montagesequenz wird als erster Schritt geprüft, ob der aktuelle Anfangszustand des Systems mit dem abgelegten Zustand der Sequenz übereinstimmt. Ist dies der Fall, kann die Sequenz ausgeführt werden. Ist dies nicht der Fall, wird versucht, aufgrund der in der Sequenz gespeicherten Auswirkung der Operationen auf die Szene herauszufinden, ob der aktuelle Zustand des Montagesystems einem Zwischenzustand der Montagesequenz entspricht. Die Montage wird vom System simulativ durchgegangen. Lässt sich ein solcher Zwischenzustand finden, der mit dem aktuellen Zustand übereinstimmt, wird ab dieser Stelle die Sequenz ausgeführt. Das Montagesystem ist damit in der Lage, eine teilweise durchgeführte Montage eines ihm bekannten Aggregates autonom zu beenden.

Beispiel

Soll z.B. der Propeller aus dem SFB-Szenario gebaut werden, so würde die in Abb. 6.15 gezeigte Sequenz ausgeführt. Es müssen zwei 3-Loch Leisten auf eine Schraube gesteckt werden, die danach in einen Schraubwürfel geschraubt wird. Der übliche Anfangszustand des Systems wären zwei Manipulatoren mit leeren Greifern. Käme in diesem Fall die Anweisung vom Instrukteur einen Propeller zu bauen, würde das System die entsprechende Sequenz komplett ausführen. Beginnt aber der Instrukteur den Bau eines Propellers mit Einzelanweisungen, erreicht er einen Zwischenzustand der entsprechenden Sequenz für den Bau eines Propellers: z.B. der eine Manipulator hält eine Schraube mit einer aufgesteckten Leiste. Würde nun die Anweisung gegeben, einen Propeller zu bauen, wäre die vollständiges Ausführung der Sequenz zum Bau eines Propeller nicht die intendierte Reaktion. Es würde ein Abschluss der begonnenen Handlung erwartet, nicht ein Neubeginn. Die in Abb. 6.16 fett markierten Instruktionen müssen nicht mehr ausgeführt werden. Durch simulatives Durchgehen der Sequenz wird die entsprechende Stelle in der Sequenz gefunden, ab der mit der Ausführung der Montagesequenz begonnen wird.

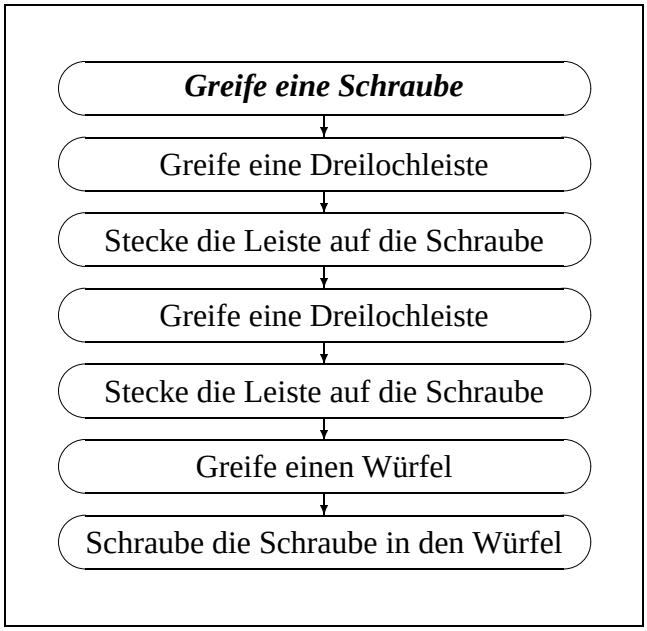


Abbildung 6.15: *Flussdiagramm der Beispielsequenz zum Bau eines Propellers.*

6.3 Modifikation

Hat das Montagesystem den Bau eines Aggregates erlernt, so sollte es ausgehend vom Gelernten auch ähnliche Aggregate zusammenbauen können. Ein erster Schritt hierzu ist die Verwendung von unterschiedlichen Bauteilen. So soll z.B. folgende Anweisung vom Instrukteur umgesetzt werden können:

Baue das Leitwerk, aber mit der blauen Schraube.

Hat das System den Bau des Leitwerks mit einer gelben Schraube, Dreilochleiste und einem gelben Schraubwürfel erlernt, so muss es jetzt die erlernte Sequenz so **modifizieren**, dass die Anweisung entsprechend umgesetzt wird; es muss eine blaue statt einer gelben Schraube gegriffen werden.

Um diese Modifikation durchzuführen, geht das Montagesystem die Sequenz in ihren unterschiedlichen Ablaufmöglichkeiten durch und sucht nach der Anweisung, die das Greifen einer Schraube realisiert. Ist diese Anweisung anhand der Instruktionen, die die interne Szenenrepräsentation modifizieren[5], gefunden, so wird der Parameter für die Farbe geändert.

[5]Siehe z.B. Abb. 6.15 die fett markierte Instruktionen.

Abbildung 6.16: *Flussdiagramm der Beispielsequenz zum Bau eines Propellers.*

Problemfälle

Die in der vorliegenden Arbeit realisierten Modifikationen gehen nicht über die Änderung vom Bauteilattributen hinaus. Aber schon diese Modifikation können bei der Umsetzung nicht trivial sein.

Mehrdeutigkeiten

Werden in einer Sequenz z.B. mehrere Schrauben verwendet, so ist nicht klar, welche Schraube ersetzt werden soll. Die Lösung ist mehrdeutig und kann nur durch den Instrukteur aufgelöst werden. In diesem Fall muss das System rückfragen und eine präzisiere Angabe verlangen oder es wird dem Instrukteur eine Auswahl angeboten. Abb. 6.17 zeigt ein Beispiel für eine solche Mehrdeutigkeit. Käme eine Anweisung, den Rumpf mit einer blauen Schraube zu montieren, so kämen beide Schrauben (fett unterlegte Instruktionen) in Frage.

Mehrdeutigkeiten mit Verzweigungen in der Handlung

Treten in der erlernten Sequenz Verzweigungen auf, die auch das Greifen des zu modifizierenden Bauteils tangiert, so kann sich das Problem noch vergrößern. Da a priori nicht gesagt werden kann, welche Handlungszweige der Sequenz durchlaufen werden, müssen alle Instruktionen, die das Bauteil greifen, modifiziert werden. Werden aber innerhalb der Sequenz mehrere gleichartige Bauteile verwendet (z.B Schrauben), so

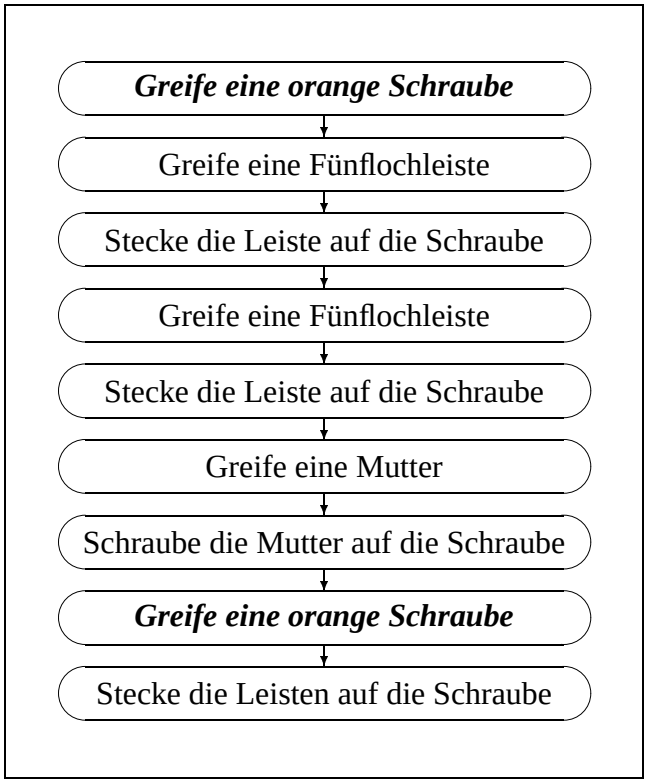


Abbildung 6.17: *Vereinfachter Ausschnitt einer Rumpfbausequenz.*

ist eine klare Zuordnung zwischen den jeweiligen Bauteilen und den Instruktionen, die greifen, nicht sichergestellt. Abb. 6.18 zeigt eine solche Sequenz. Es wird sowohl in der rechten wie auch in der linken Verzweigung ein Teil des Rumpfes gebaut. Es wird aber jeweils mit der anderen Schraube angefangen. Soll nun eine Schraube gegen eine andere ausgetauscht werden (gelb gegen blau), so muss im rechten Handlungszweig die erste oder die zweite und im linken die zweite resp. die erste Greifinstruktion geändert werden. Dies kann so aber nicht aus der Sequenz ersehen werden. Dafür ist es notwendig, die Montage simulativ durchzugehen, um beurteilen zu können, wo ein gegriffenes Bauteil im Aggregat verbaut wird.

6.4 Kognitive Aspekte

[Eys94] Die Bezeichnung Gedächtnissysteme bezeichnet vermeintliche Verhaltens- und Erkennungssysteme des Gehirns verbunden mit verschiedenen Arten des Lernens und Erinnerns. *Gedächtnis* ist eine allgemeine Bezeichnung für verschiedene Arten von Erwerb, Aufbereitung und Gebrauch von Informationen, Fertigkeiten und Wissen. Da es Hinweise auf die Existenz verschiedener neuronaler Teilbereiche für verschiedene Hirntätigkeiten gibt, neigt man dazu, mehr den verschieden Gedächtnisprozessen auch

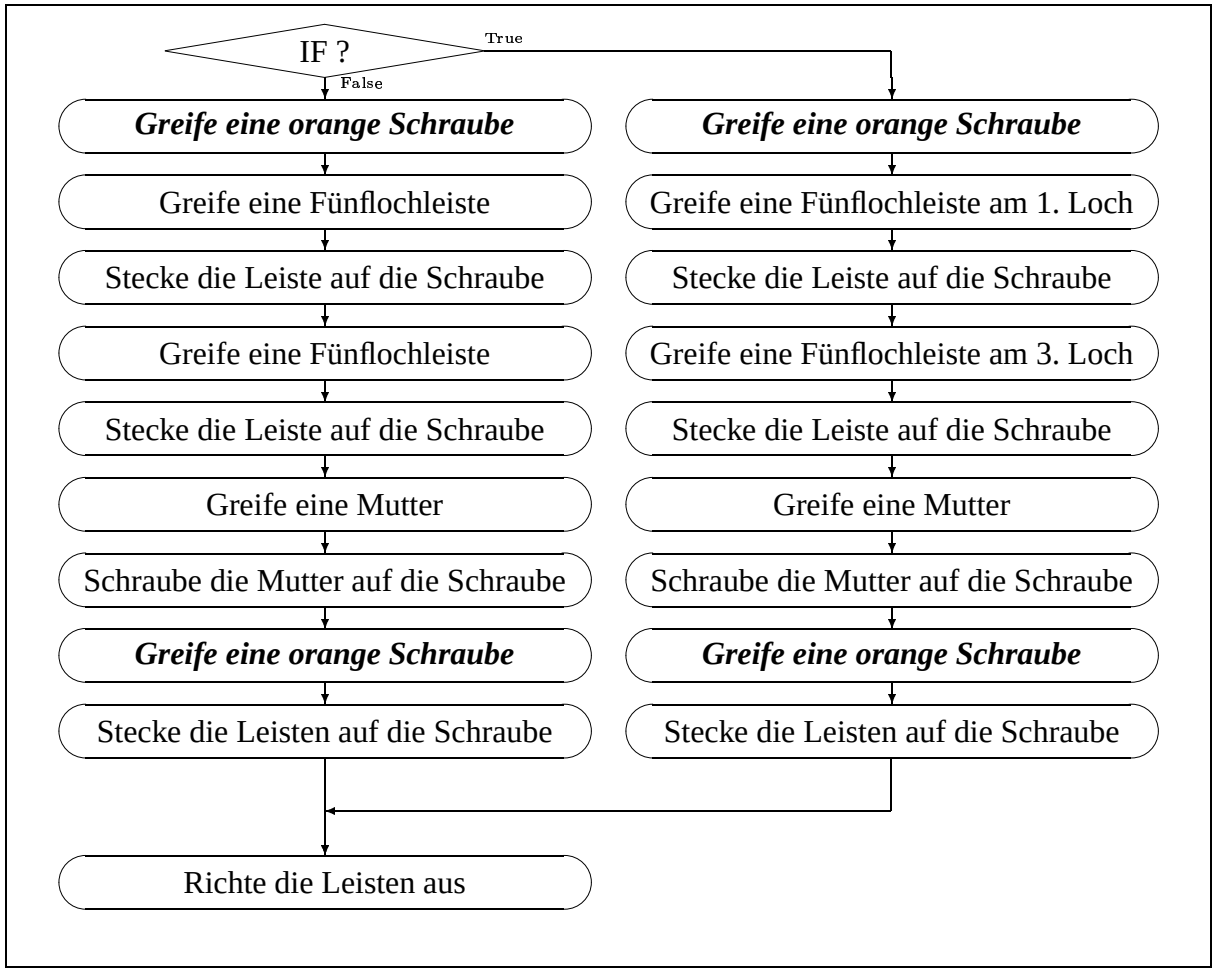


Abbildung 6.18: *Vereinfachter Ausschnitt einer Rumpfbausequenz mit Verzweigung der Handlung.*

verschiedenen Gedächtnissystemen zuzuordnen [Wei87]. Alle Systeme haben gemeinsam, dass sie Informationen behalten und sie für späteres Verhalten und kognitive Funktionen zur Verfügung stellen, die auf früherem Verhalten und Erfahrungen beruhen. Sie unterscheiden sich in der Art, wie sie die Information handhaben und in ihrer Funktion.

Der Großteil der Forschung stimmt der Klassifikation der Form von Lernen und Gedächtnis in drei hypothetische Systemen zu: episodisches Gedächtnis, semantisches Gedächtnis und prozedurales Gedächtnis [Tul85]. Diese Systeme werden hier betrachtet.

Episodisches Gedächtnis ist ein Gedächtnissystem, welches es einer Person ermöglicht, sich subjektiv an konkrete persönliche Episoden oder Ereignisse und deren zeitliche Relationen zueinander zu erinnern [Tul83]. Episodisches Gedächtnis ist in seinem Wesen ein *mentales* Phänomen. Daher basieren sowohl Erinnerung als

auch Planung auf dem episodischen Gedächtnis [TKC+94].

Semantisches Gedächtnis: Der Inhalt des semantischen Gedächtnis lässt sich als *allgemeines Weltwissen* beschreiben [Qui66]. Das semantisches Gedächtnis ist anfangs eng mit dem Wissen, das sich durch Sprache beschreiben lässt, verbunden worden, wird jetzt aber breiter und differenzierter aufgefasst. Der Inhalt des semantischen Gedächtnises muss nicht einen persönlichen oder zeitlichen Bezug haben.

Das prozedurale Gedächtnis wird als ein großes System verstanden, dass schon in einem frühen Stadium der Evolution gebildet wurde und das die meisten lebenden Organismen besitzen. Prozedurales Gedächtnis ermöglicht es Organismen, die Verbindung zwischen Stimulus und Reaktion zu erkennen. Die Information im prozeduralen Gedächtnis ist nicht symbolisch und wird als *Verhalten* bezeichnet. Die Antwort, die durch das prozedurales Gedächtnis aufgrund eines Stimulus erzeugt wird, ist nicht abhängig von einer bestimmte äußeren Situation. Das prozedurale Gedächtnis bedient sich auch keines *Wissens* über die Welt. Im Gegensatz zu Informationen aus dem semantischen und episodischen Gedächtnis, deren Informationen sehr schnell aufgebaut werde können, erfolgt der Aufbau des prozeduralen Gedächtnises eher langsam.

Wird das vorgestellte Lernverfahren unter den oben beschriebenen Aspekten betrachtet, so finden sich Verbindungen. *Semantisches* Wissen ist notwendig, um die Anweisungen des Instrukteur zu verstehen und interpretieren zu können. Die Anweisungen *Greifen, Ablegen, Stecken und Schrauben* erfordern vom Montagesystem weitergehende Verarbeitungschritte. So muss die Anweisung auf Plausibilität (ist die Anweisung an sich vernünftig) und auf Machbarkeit (ist die Anweisung durchführbar) geprüft werden. Zusätzlich lassen die Anweisungen dem Montagesystem gewisse Interpretationsfreiheiten (z.B. Disambiguierung von Bauteilen), die vom System erst präzisiert werden müssen, bevor auf *prozedurales* Wissen zurückgegriffen und die Anweisungen umgesetzt werden. Zu allen diesen Verarbeitungsschritten ist semantisches Wissen notwendig und ist daher im kleinen oder großen Umfang in jedem nicht trivialen Programm vorhanden.

Wird von dem Montagesystem verlangt, Montagevorgänge zu einem späteren Zeitpunkt selbstständig zu wiederholen, so ist der Aufbau von *episodischen* Wissen erforderlich. Der erste dazu erforderliche Schritt ist die Speicherung der Reihenfolge der zu tätigen Montageoperation. Das hier vorgestellt Montagesystem ist in der Lage, sich an konkrete Episoden und durch Ablegen der anliegenden Sensordaten an Ereignisse zu *erinnern* und diese zeitlich in Bezug setzen zu können. Zusätzlich ist es durch den Aufbau einer intern symbolischen Repräsentation der Szene in der Lage, Montagevorgänge *mental* durchzuführen und dadurch Entscheidungen zu fällen. Die gespeicherte Information wird nicht nur gespeichert, sondern weiterverarbeitet und kann zu einem späteren Zeitpunkt abgerufen werden. Ein Informationstransfer ist in engen Grenzen möglich.

Nach der Fusion von verschiedenen Montagebeispiele für ein Aggregat ist neues *prozedurales* Wissen aufgebaut worden, dessen sich das Montagesystem in nachfolgenden Montagevorgängen bedienen kann. Während der Wiederholung der gelernten Montagesequenz wird nicht über das Montageziel oder den aktuellen Montagezustand reflektiert, sondern es werden die entsprechenden Montageoperation der Reihe nach abgearbeitet. Handlungen werden aufgrund eines Stimulus (anliegende Sensordaten) durchgeführt. Wie auch beim menschlichen Vorbild dauert auch bei hier vorgestellten System der Aufbau des prozeduralen Wissen länger. Dies kommt durch die Verwendung von statistischen Methoden zur Sensorauswertung.

Semantisches Wissen wird in der vorliegenden Arbeit nicht bzw. nur im sehr geringen Maßen aufgebaut. Das Montagesystem besitzt nach dem Erlernen einer Montagesequenz zwar Wissen über den Aufbau eines Aggregates, verwendet dieses Wissen aber zu einem späteren Zeitpunkt nicht mehr.

6.5 Zusammenfassung

In diesem Kapitel werden zwei Arten der Generalisierung von Montagesequenzen behandelt:

1. Das Zusammenführen von unterschiedlichen Beispielesequenzen einer Montage zu einer generell verwendbaren Montageanleitung.
2. Die Anwendung von bestehenden gelernten Montageanleitungen mit Variierung der verwendeten Bauteile.

Existieren für den Bau eines Aggregates mehrere Beispielsequenzen, so werden diese nach Abschluss des Lernvorganges fusioniert. Sind die zu fusionierenden Sequenzen in Teilbereichen nicht identisch, so werden Verzweigungen in die Zielsequenz eingebaut. Hierbei werden drei unterschiedliche Fälle unterschieden:

1. Die Quellsequenz enthält gegenüber der Zielsequenz zusätzliche Instruktionen.
2. Die Zielsequenz enthält gegenüber der Quellsequenz zusätzliche Instruktionen.
3. In der Quell- und Zielsequenz existieren unterschiedliche Teilsequenzen.

Die boolsche Funktion dieser Verzweigung wird anhand der gespeicherten Sensordaten ermittelt und über den in Kapitel 4 vorgestellten Ansatz realisiert.

Bei der Wiederholung einer gelernten Sequenz wird vor der Ausführung überprüft, ob der Zustand des Montagesystem mit dem gelernten Startzustand übereinstimmt. Ist dies nicht der Fall, so wird mit Hilfe der virtuellen Durchführung der Montage der aktuelle Zustand in den Zwischenzuständen der Sequenz gesucht. Kann ein solcher Zustand ermittelt werden, so wird die Sequenz ab entsprechender Stelle ausgeführt.

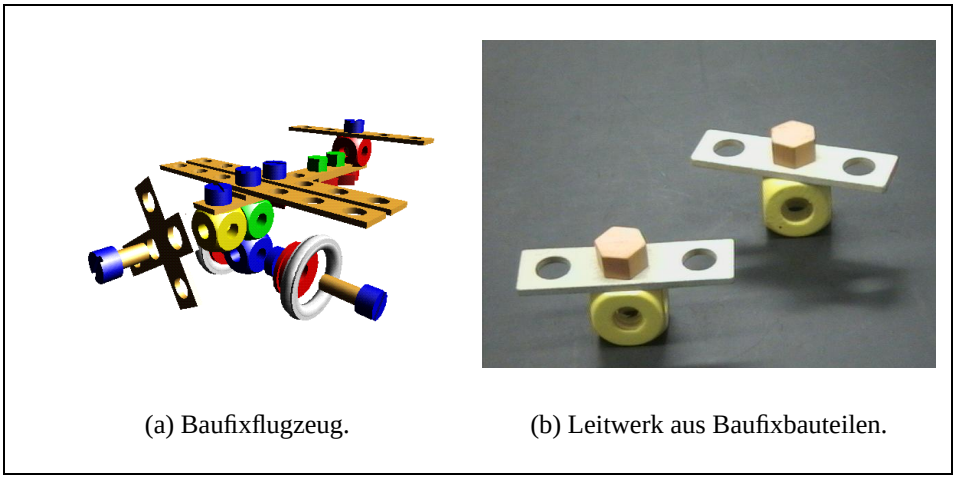
Durch die Modifikation der Parameter für eine Greifoperation ist das Montagesystem in der Lage, die gelernten Aggregate auch leicht abgewandelt bauen zu können.

Kapitel 7

Gesamtbeispiel

7.1 Zielaggregate

Zur Demonstration der Funktionalität des System wird der Bau zweier Teilaggregate eines Flugzeugs (Abb.7.1 (a)) mit Teilen des Spielzeuges *Baufix* durchgeführt. Das erste Beispiel ist der Bau eines Leitwerks (Abb. 7.1 (b)) und besteht aus einer Schraube, einer Leiste und einem Schraubwürfel. Der Bau dieses Aggregates ist nicht besonders



(a) Baufixflugzeug. (b) Leitwerk aus Baufixbauteilen.

Abbildung 7.1: *Aggregate aus dem Spielzeug* Baufix.

komplex, aber es wird die Montage einmal mit Roboter 0 und einmal mit Roboter 1 begonnen. Das andere Beispielaggregat ist der Rumpf des Flugzeugs (Abb. 7.2). Dieser besteht aus zwei versetzen Fünflochleisten die mit zwei Schrauben und Rautenmuttern zusammengeschraubt sind. Im Gegensatz zum Leitwerk müssen die beiden Leisten aneinander ausgerichtet und zum Schrauben arretiert werden.

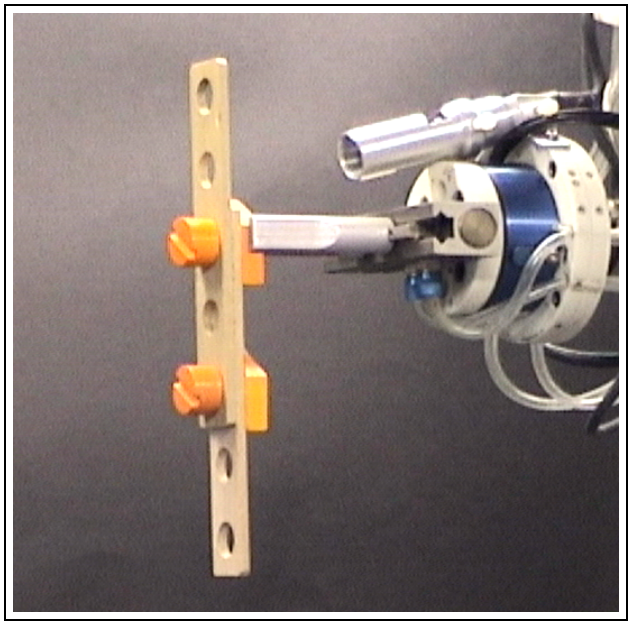


Abbildung 7.2: *Rumpf des Baufix-Flugzeuges.*

7.2 Verwendete Module und Skripte

Folgende Module und Skripte sind zum Bau der Aggregate verwendet worden:

Module

OpenHand: Öffnet die Hand eines Manipulators. Parameter dieses Moduls ist die Nummer des Roboters.

MoveARel: Bewegt einen Manipulator relativ entlang des Toolannäherungsvektors. Die Parameter für dieses Modul sind die Länge der Strecke, die Manipulatorgeschwindigkeit und die Kraft, bei der die Bewegung abgebrochen werden soll.

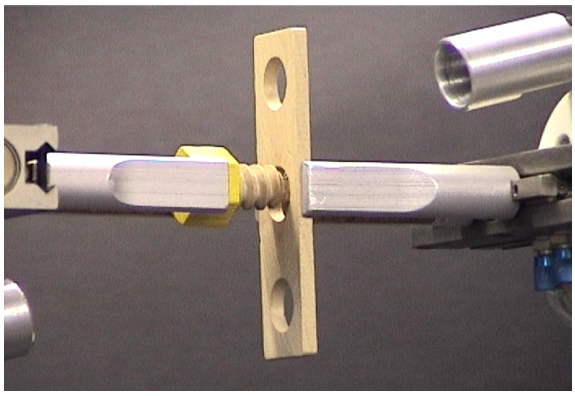
MoveRel: Bewegt den Manipulator relativ entlang seiner Toolachsen. Die Parameter für dieses Modul sind die Nummer des Manipulators, die Bewegungsgeschwindigkeit und die Länge der Strecken.

Zeroing: Dreht das sechste Gelenk eines Manipulators so nah wie möglich an den Winkelwert Null. Als Parameter wird die Nummer des Manipulators und die Schrittweite in Grad angegeben, mit dem das Gelenk gedreht werden darf.

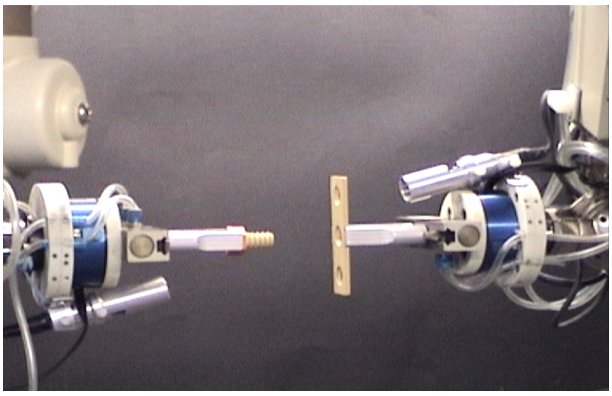
Move: Bewegt einen Manipulator an eine bestimmte Position. Parameter sind die Nummer des Manipulators und die Position. Die Position kann auf zwei verschiedene Arten angegeben werden: direkt als Ort und Orientierung oder als Referenz auf eine Position, die im *Blackboard* gespeichert ist.

Store Position: Speichert die Position eines Manipulators in einer Variablen im Blackboard.

Rotate: Dreht das sechste Gelenk eines Manipulators um einen angegebenen Winkel.



(a) Schraubenspitze über dem Loch einer Leiste.



(b) Rendezvousposition der Roboter.

Abbildung 7.3: *Verschiedene Zustände während der Montage.*

Skripte

GraspPart: Greift ein Bauteil vom Montagetisch. Dieses Skript wird der Typ (Schraube, Leiste, ...) und nähere beschreibende Angaben (Farbe, Anzahl der Löcher, Greifpunkt) des zu greifenden Bauteils als Parameter übergeben.

Rendezvous: Bewegt die beiden Endeffektoren der Manipulatoren in eine Gegenüberposition (Abb. 7.3 (b)).

PlugIn: Bewegt einen Manipulator durch gefügige Bewegungen in das Loch einer Leiste. Parameter sind die Nummern des steckenden und des haltenden Roboters. Für die Operation wird zuerst die Schraubenspitze auf die Oberfläche der Leiste positioniert und durch eine kreisförmige Bewegung das Loch in der Leiste gesucht, so dass die Schraubenspitze sich über den Loch befinden (Abb. 7.3 (a)). Danach wird die Schraube in das Loch gesteckt.

Unlink: Zieht die Arme beider Manipulatoren entlang der jeweiligen Annäherungsachse zurück.

Screw: Schraubt eine Schraube in ein Gewinde. Parameter sind die Nummern des schraubenden, des haltenden Roboters und des Anzugsmomentes.

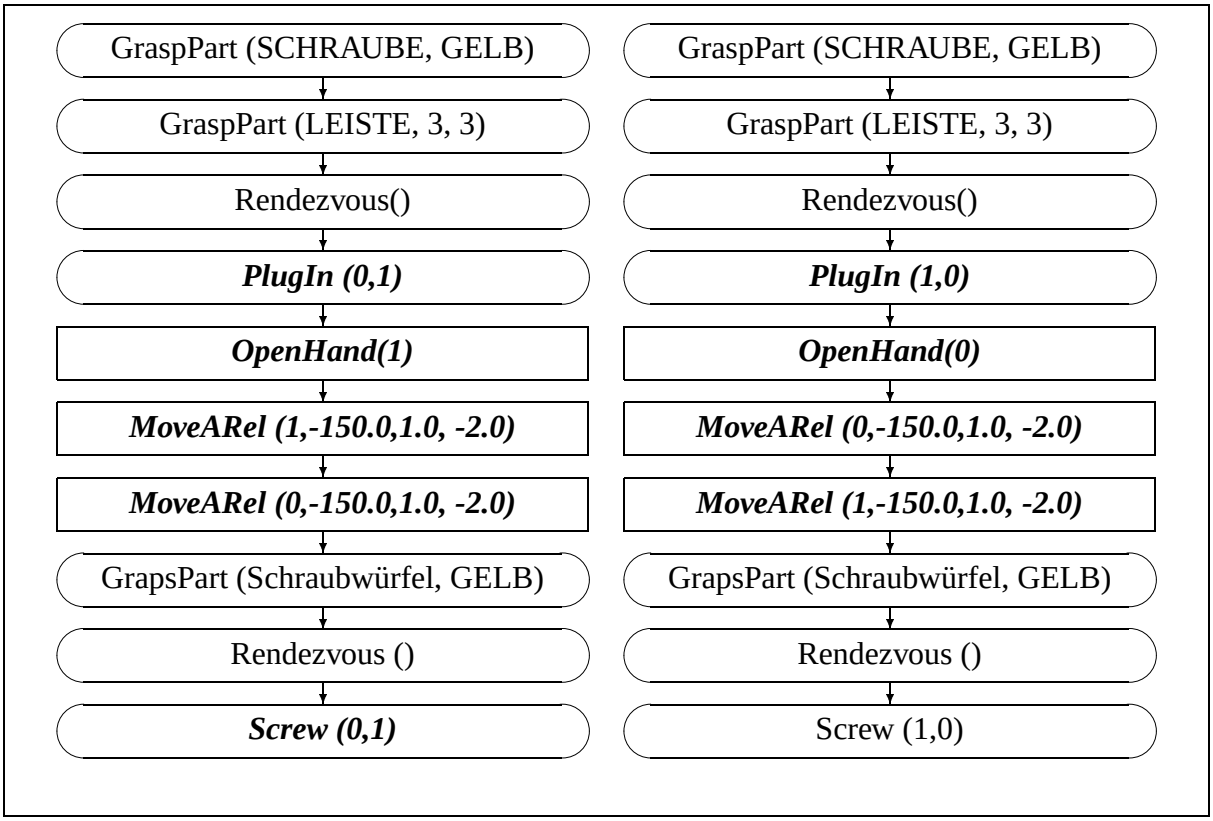


Abbildung 7.4: *Flussdiagramm zweier Sequenzen zum Bau des Leitwerks. Die Instruktionen zur Aktualisierung der internen Repräsentation sind zur besseren Übersicht in den Flussdiagrammen nicht dargestellt.*

AlignRight: Führt eine Ausrichtbewegung entlang der rechten Seite des anderen Manipulators durch.

AlignLeft: Führt eine Ausrichtbewegung entlang der linken Seite des anderen Manipulators durch.

Die Skripte repräsentieren die zur Verfügung stehenden komplexen Bewegungsanweisungen, die Module direkte. Eine Anweisung des Instrukteurs wird daher von *OPERA* in einen Skriptaufruf umgesetzt und vom Skriptinterpreter ausgeführt. Bei erfolgreicher Durchführung der Anweisung wird der Modul- oder Skriptaufruf in einem Skript zusammen mit den vor der Ausführung gültigen Sensordaten abgespeichert (siehe Abb. 7.5). Nach Beendigung der Montage stellt dieses Skript eine Sequenz von Montageanweisungen ein Beispiel für eine mögliche Montage dar und wird vom Montagesystem mit den schon existierenden Beispielsequenzen zusammengefasst. Sowohl die Entgegennahme der Anweisungen, das Prüfen, die Ausführen der Anweisungen, als auch das Zusammenfassen der Beispielsequenzen wird durch entsprechende Module übernommen, die die jeweiligen Informationen zusammenfassen.

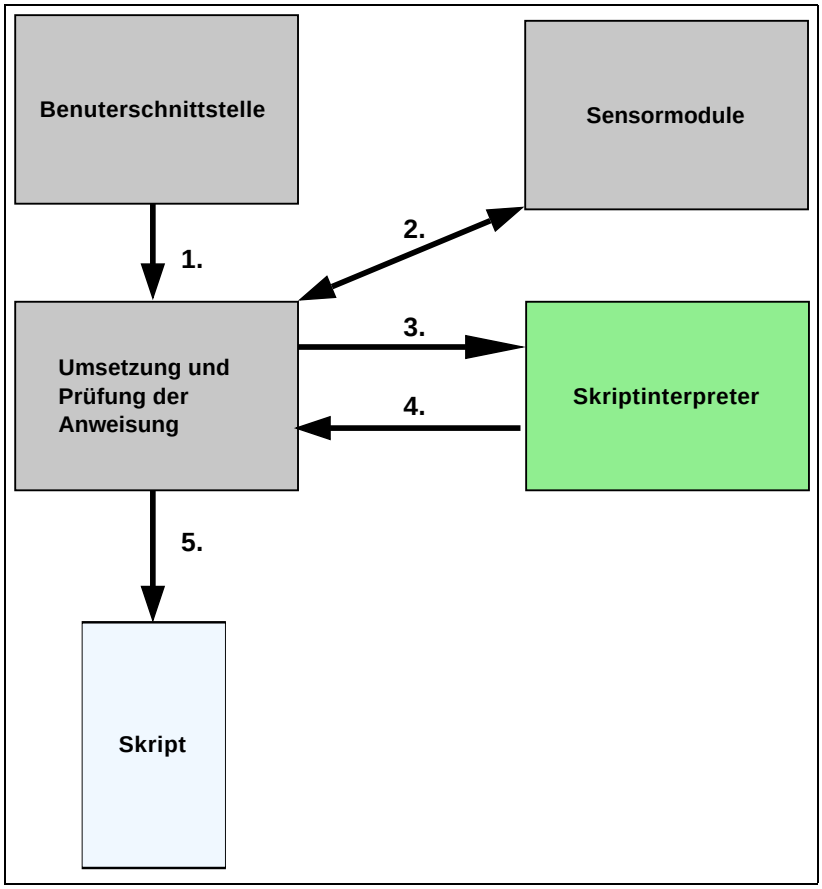


Abbildung 7.5: *Reihenfolge der Bearbeitung einer Instrukteursanweisung. (1) Entgegennahme der Anweisung über die Benutzerschnittstelle und Umsetzen in einen Skriptaufruf. (2) Speicherung der aktuellen Sensordaten. (3) Übergabe des Skriptes an den Skriptinterpreter zur Ausführung. (4) Rückmeldung des Skriptinterpreter über den Erfolg der Ausführung und (5) Speicherung der Skriptaufrufen mit Sensordaten im Erfolgsfall.*

7.3 Bau des Leitwerks

Zu Bau des Leitwerks müssen folgende Aktionen vom Montagesystem durchgeführt werden:

1. Greifen einer Schraube und einer Leiste;
2. Aufstecken der Leiste auf die Schraube;
3. Greifen eines Schraubwürfels und
4. Schrauben der Schraube in den Schraubwürfel.

Wird das Montagesystem über die oben beschriebenen Instruktionen angeleitet, ein Leitwerk zu bauen, ergeben sich zwei verschiedene Sequenzen (Abb. 7.4), die sich in

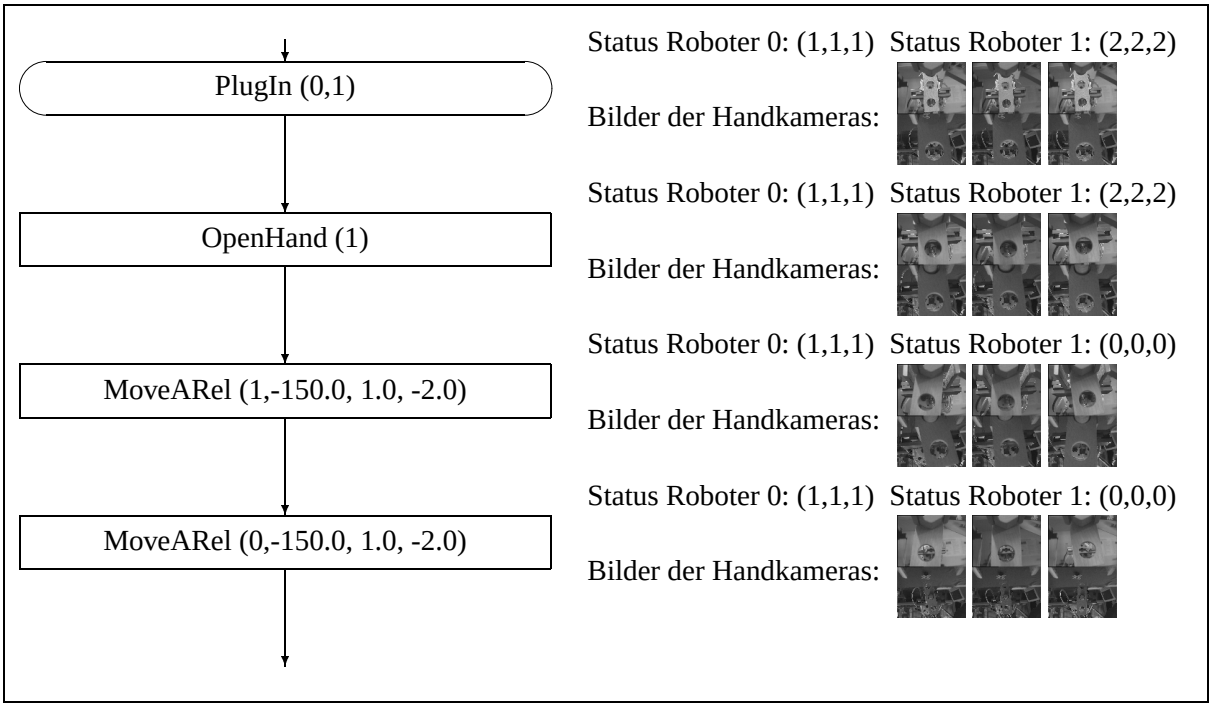


Abbildung 7.6: *Flussdiagramm der 1. Teilsequenz (Leitwerk) mit Sensordaten. Die Instruktionen zur Aktualisierung der internen Repräsentation sind zur besseren Übersicht in den Flussdiagrammen nicht dargestellt.*

den (fett) markierten Instruktionen $7 - 14$ und $19 - 20$ unterscheiden. Die Unterschiede kommen dadurch zustande, dass im ersten Fall (Abb. 7.4 (a)) Roboter 0 die Schraube greift und im zweiten (Abb. 7.4 (b)) Roboter 1. Dies führt zu unterschiedlichen Parametern bei den Instruktionen: `PlugIn, OpenHand` und `MoveARel`. Werden die zu den Instruktionen aufgezeichneten Sensorinformationen betrachtet, lassen sich die Unterschiede auch in den Sensordaten wiederfinden (Abb. 7.6) und (Abb. 7.7). Die Abbildungen zeigen den ersten unterschiedlichen Teil der beiden Sequenzen mit den jeweiligen zu den Instruktionen aufgenommenen Sensordaten. Übersichtshalber werden nicht alle Sensordaten gezeigt. Es sind nur diejenigen Daten dargestellt, in denen sich auch Unterschiede zwischen den beiden Sequenzen zeigen. Dies sind die fusionierten Bilder der Handkameras und die Angabe, welches Bauteil die Roboter in der Hand halten. Ein Wert von eins bezeichnet eine Schraube und eine Wert von zwei eine Leiste. Da für jeden Fall nicht nur eine Beispielsequenz vom System gespeichert wurden, sondern mehrere, existieren für eine Instruktion mehrere Sensordatenbeispiele von denen jeweils drei abgebildet sind.

Sollen beide Sequenzen zusammengeführt werden, so muss eine Verzweigung gemäß dem in Kapitel 6 vorgestellten Verfahren erzeugt werden. Abb. 7.10 zeigt das Resultat der Zusammenführung der beiden Sequenzen. Es sind zwei Verzweigungen

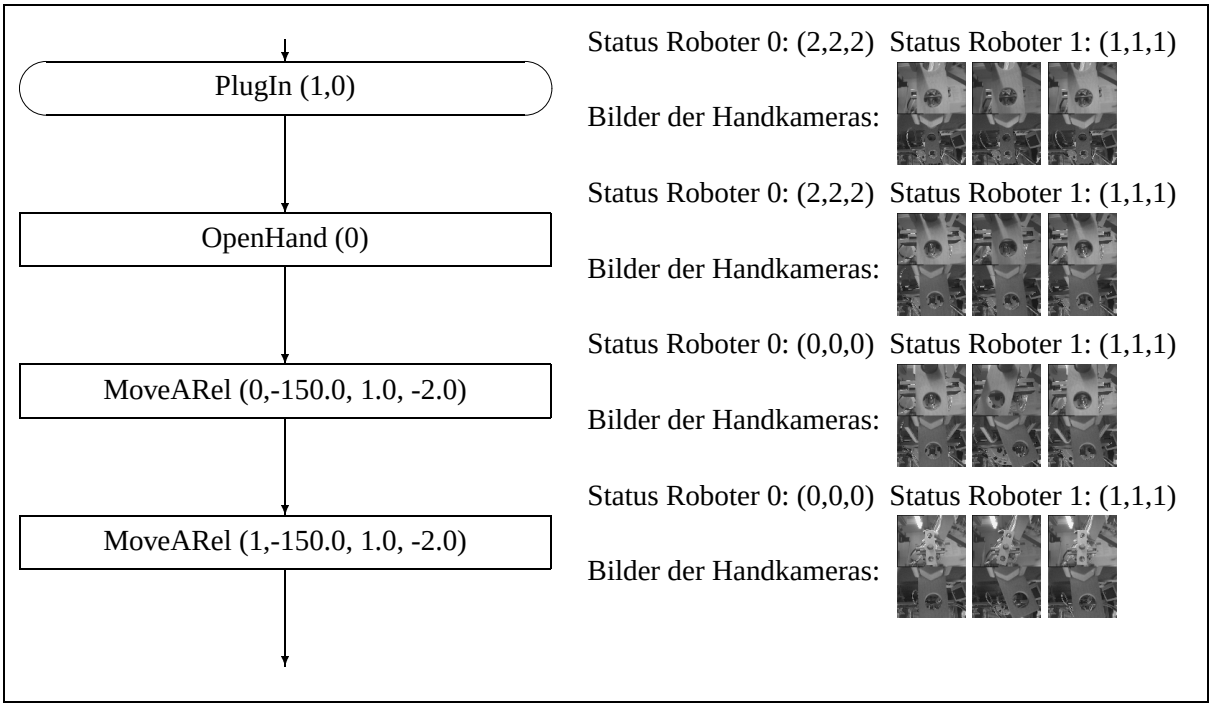


Abbildung 7.7: *Flussdiagramm 2. Teilsequenz (Leitwerk) mit Sensordaten. Die Instruktionen zur Aktualisierung der internen Repräsentation sind zur besseren Übersicht in den Flussdiagrammen nicht dargestellt.*

(`Verzweigung 1` und `Verzweigung 2`) eingefügt worden.

Ermittlung der Verzweigungsbedingungen

Es wird zwischen internen und externen Sensoren bzw. Sensordaten unterschieden. Interne Sensorendaten werden aus der Robotersteuerung und der internen Repräsentation gewonnen; externe Sensorendaten kommen von externen Sensoren, wie z.B. Kraftmomentensensor, Kameras, usw. . Insgesamt existieren neun Beispiele, so dass die Aufstellung der Sensordaten zur Berechnung der ersten Verzweigung die folgende Tabelle 7.1 ergibt. Die einzelnen Sensoren haben folgende Bedeutung:

Handkameras: Fusionierte Bilder der beiden Handkameras der Roboter (externer Sensor).

Status beschreibt, was die Manipulatoren in den Greifern halten. Der Wert Null korrespondiert mit einem leeren Greifer. Die Werte eins bis vier stehen für Schraube, Leiste, Schraubwürfel und Mutter (interner Sensor).

Schwerpunkt: Ist dieser Wert eins, so befindet sich der Schwerpunkt des Teilaggregat oberhalb der Greiferbacken (interner Sensor).

Sensor	Bsp. 1	Bsp. 2	Bsp. 3	Bsp. 4	Bsp. 5	Bsp. 6	Bsp. 7	Bsp. 8	Bsp. 9
Handkameras									
Status Roboter 0	1	1	1	1	1	1	2	2	2
Status Roboter 1	2	2	2	2	2	2	1	1	1
Status Greifer 0	0	0	0	0	0	0	0	0	0
Status Greifer 1	0	0	0	0	0	0	0	0	0
Schwerpunkt Roboter 0	0	0	0	0	0	0	0	0	0
Schwerpunkt Roboter 1	0	0	0	0	0	0	0	0	0
Stabilität Roboter 0	1	1	1	1	1	1	1	1	1
Stabilität Roboter 1	1	1	1	1	1	1	1	1	1

Tabelle 7.1: Sensordaten an der Verzweigung.

Stabilität: Ist der Wert eins, so ist das gehaltene Teilaggregat stabil, sonst ist der Wert Null (interner Sensor).

Die Berechnung der Korrelation zwischen den Sensordaten und der Sollausgabe des Reglers sind in Tabelle 7.2 dargestellt. Da der Betrag der Korrelation mindestens 0.8

Sensor	Korrelation
Handkameras	0.942794
Status Roboter 0	0.816497
Status Roboter 1	0.6860

Tabelle 7.2: Korrelation der relevanten Sensordaten zur jeweiligen Verzweigung (1. Verzweigung).

betragen muss, damit ein Sensor als Eingangsgröße für den nachgeschalteten B-Spline Regler verwendet wird, ist die Dimension des trainierten Reglers zwei. Abb. 7.8 zeigt die Ausgabefunktion des Reglers nach dem Training, dessen Ausgaben im Intervall $[0, 1]$ liegen. Alle Werte kleiner als 0.5 führen dazu, dass die Bedingung für die Verzweigung **nicht** erfüllt ist, also keine Verzweigung stattfindet. Alle Werte größer gleich 0.5 führen zu einer Verzweigung.

Wie erwartet, lässt sich anhand der gegriffenen Bauteile und der Handkameras entscheiden, welcher Teil der Sequenz ausgeführt werden muss. Die Ermittlung und Berechnung des Reglers für die zweite Verzweigung erfolgte analog. Es ergeben sich für die berücksichtigten Sensordaten die in Tabelle 7.3 gezeigten Korrelationen. Zusätzlich zur ersten Verzweigung wird hier die Stabilität des Teilaggregates (Schraube mit aufgesteckter Leiste) betrachtet. Je nachdem mit welchem Roboter die Montage begonnen wurde, befindet sich das Aggregat im Greifer des einen oder des anderen Manipulators, so dass sich auch mit Hilfe dieses Sensors entschieden werden kann, welche Teilsequenz ausgeführt werden muss.

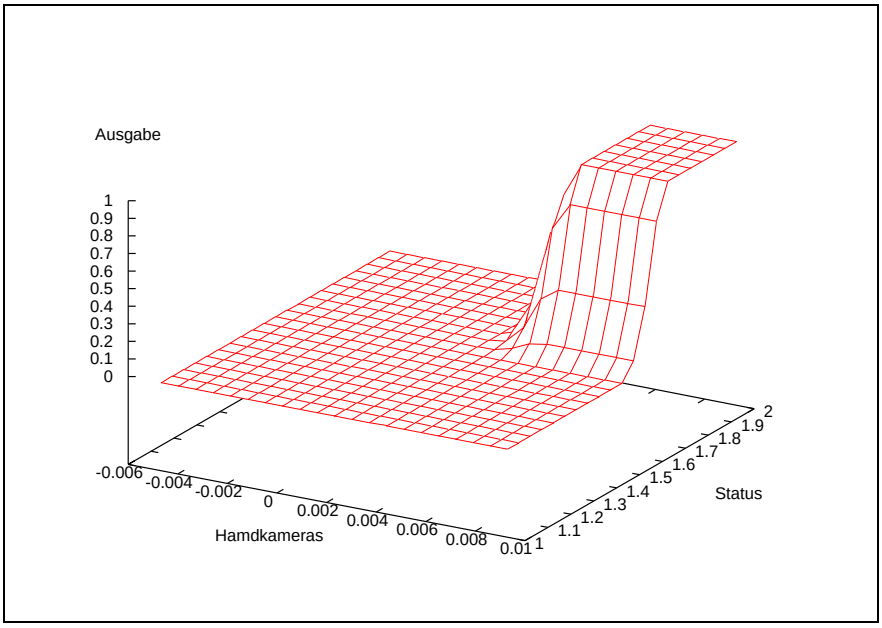


Abbildung 7.8: *Ausgabefunktion des trainierten B-Spline Fuzzy Reglers.*

Sensor	Korrelation
Handkameras	0.942803
Status Roboter 0	0.816497
Stabilität Roboter 0	0.816497

Tabelle 7.3: Korrelation der relevanten Sensordaten zur jeweiligen Verzweigung (2. Verzweigung).

7.4 Bau des Rumpfes

Der Bau des Rumpfes (siehe Abb. 7.2) kann mit den denselben Operation durchgeführt werden, die zum Bau des Leitwerk verwendet werden. Im Unterschied zum Leitwerk müssen die Leisten zueinander ausgerichtet werden, damit beide mit einer zweiten Schraube verbunden werden können. Da beim Schrauben die mittig aufgesteckte Leiste verdreht wird, ist diese Operation notwendig. Damit das Ausrichten im jeden Fall erfolgreich ist, muss, je nach Winkel der Leiste, eine andere Trajektorie mit dem ausrichtenden Manipulator abgefahren werden. Beispielsequenzen für den Bau des Rumpfes, werden sich daher auf jeden Fall an diesem Punkt unterscheiden. Der Winkel der Leiste ist nicht durch interne Zustandsvariablen wie .z.B `Status Roboter 0` beim Leitwerk zu ermitteln. Daher müssen auf jeden Fall externe Sensoren (Handkameras, Kraftmomentensensoren), die notwendigen Informationen liefern. Wie schon in Kapitel 6.2.3 gezeigt, ist die notwendige Information über die Handkameras zu gewinnen.

Für den Bau den Rumpfes müssen folgende Montageoperationen durchgeführt wer-

den:

1. Greifen einer Schraube und einer Fünflochleiste in der Mitte;
2. Aufstecken der Leiste auf die Schraube;
3. Greifen einer Fünflochleiste am ersten Loch;
4. Aufstecken der Leiste auf die Schraube;
5. Greifen einer Mutter und Verschrauben mit der Schraube;
6. Ausrichten der ersten Leiste;
7. Arretieren der Leisten und Festdrehen der Schraube;
8. Umgreifen des Teilaggregates;
9. Greifen einer Schraube und Aufstecken des Teilaggregates;
10. Greifen einer Mutter und Verschrauben mit der Schraube.

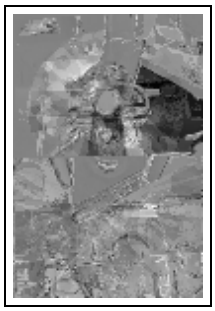


Abbildung 7.9: *Visualisierung des Projektionsvektors.*

Es wurden insgesamt 25 Beispielsequenzen erstellt, deren Fusion die in Abb. 7.11 gezeigt Sequenz ergibt. Zur besseren Übersicht wurde im Gegensatz zum Bau des Leitwerk die Montage immer mit demselben Manipulator (mit der Nummer 0) begonnen. Es ergibt sich daher nur eine Verzweigung, an der entschieden wird, welche Trajektorie zum Ausrichten der Leisten mit dem Manipulator abgefahren werden muss. Folgende Sensordaten wurden aufgezeichnet:

Handkameras: Fusionierte Bilder der beiden Handkameras der Roboter (externer Sensor).

Status beschreibt, was die Manipulatoren in den Greifern halten. Der Wert Null korrespondiert mit einem leeren Greifer. Die Werte eins bis vier stehen für Schraube, Leiste, Schraubwürfel und Mutter (interner Sensor).

Stabilität: Ist der Wert eins, so ist das gehaltene Teilaggregat stabil, sonst ist der Wert Null (interner Sensor).

Kraft bezeichnet die Kraft in Newton entlang der N-, O- oder A-Achse des Manipulatortools (externer Sensor).

Moment: Dies ist das Drehmoment in Newtoncentimeter um die N-, O- oder A-Achse des Manipulatortools (externer Sensor).

Status Greifer gibt an, ob der Greifer des angegeben Manipulators geschlossen oder offen ist (interner Sensor).

Der Entscheidungssollwert gibt an, ob der Regler anhand dieser Daten eine Null oder eine Eins ausgeben muss. Die folgenden Tabellen zeigen die aufgezeichneten Sensordaten, welche vor dem Ausrichtvorgang aufgezeichnet wurden.

Sensor	Bsp. 1	Bsp. 2	Bsp. 3	Bsp. 4	Bsp. 5	Bsp. 6	Bsp. 7	Bsp. 8	Bsp. 9
Handkameras									
Stabilität Roboter 0	1	1	1	1	1	1	1	1	1
Status Roboter 0	1	1	1	1	1	1	1	1	1
Stabilität Roboter 1	1	1	1	1	1	1	1	1	1
Status Roboter 1	0	0	0	0	0	0	0	1	1
Kraft Roboter 0 N	0	0	0	0	0	0	0	0	0
Kraft Roboter 0 O	0	0	0	0	0	0	0	0	0
Kraft Roboter 0 A	0	0	0	0	0	0	0	0	0
Moment Roboter 0 N	0	0	0	0	0	0	0	0	0
Moment Roboter 0 O	0	0	0	0	0	0	0	0	0
Moment Roboter 0 A	0	0	0	0	0	0	0	0	0
Kraft Roboter 1 N	0	0	0	0	0	0	0	0	0
Kraft Roboter 1 O	0	0	0	0	0	0	0	0	0
Kraft Roboter 1 A	0	-0.5	0	-0.5	0	0	0	0	0
Moment Roboter 1 N	0	0	0	0	0	0	0	0	0
Moment Roboter 1 O	0	0	0	0	0	0	0	0	0
Moment Roboter 1 A	0	0	0	0	0	0	0	0	0
Status Greifer 0	0	0	0	0	0	0	0	0	0
Status Greifer 1	0	0	0	0	0	0	0	0	0
Entscheidungssollwert	0	0	0	0	0	0	0	0	0

Es ist leicht zu erkennen, dass anhand der internen Sensoren keine Entscheidung abgeleitet werden kann. Die Ausgangsituation ist in der internen Repräsentation immer dieselbe.

Sensor	Bsp. 10	Bsp. 11	Bsp. 12	Bsp. 13	Bsp. 14	Bsp. 15	Bsp. 16	Bsp. 17	Bsp. 18
Handkameras									
Stabilität Roboter 0	1	1	1	1	1	1	1	1	1
Status Roboter 0	1	1	1	1	1	1	1	1	1
Stabilität Roboter 1	1	1	1	1	1	1	1	1	1
Status Roboter 1	0	0	0	0	0	0	0	1	1
Kraft Roboter 0 N	0	0	0	0	0	0	0	0	0
Kraft Roboter 0 O	0	0	0	0	0	0	0	0	0
Kraft Roboter 0 A	0	0	0	0	0	0	0	0	0
Moment Roboter 0 N	0	0	0	0	0	0	0	0	0
Moment Roboter 0 O	0	0	0	0	0	0	0	0	0
Moment Roboter 0 A	0	0	0	0.5	0.5	0	0	0	0
Kraft Roboter 1 N	0	0	0	0	0	0	0	0	0
Kraft Roboter 1 O	0	0	0	0	0	0	0	0	0
Kraft Roboter 1 A	0-0.5	0	0	0	-0.5	-0.5	0	0	0
Moment Roboter 1 N	0	0	0	0	0	0	0	0	0
Moment Roboter 1 O	0	0	0	0	0	0	0	0	0
Moment Roboter 1 A	0	0	0	0	0	0	0	0	0
Status Greifer 0	0	0	0	0	0	0	0	0	0
Status Greifer 1	0	0	0	0	0	0	0	0	0
Entscheidungssollwert	0	0	0	1	1	1	1	1	1

Nur bei den externen Sensoren sind Unterschiede zu erkennen. Folglich werden auch nur die Daten der Sensoren `Handkameras`, `Kraft Roboter 0 A` und `Kraft Roboter 1 A` im ersten Arbeitsschritt als mögliche Entscheidungsquellen ermittelt.

Sensor	Bsp. 19	Bsp. 20	Bsp. 21	Bsp. 22	Bsp. 23	Bsp. 24	Bsp. 25
Handkameras							
Stabilität Roboter 0	1	1	1	1	1	1	1
Status Roboter 0	1	1	1	1	1	1	1
Stabilität Roboter 1	1	1	1	1	1	1	1
Status Roboter 1	0	0	0	0	0	0	0
Kraft Roboter 0 N	0	0	0	0	0	0	0
Kraft Roboter 0 O	0	0	0	0	0	0	0
Kraft Roboter 0 A	0	0	0	0	0	0	0
Moment Roboter 0 N	0	0	0	0	0	0	0
Moment Roboter 0 O	0	0	0	0	0	0	0
Moment Roboter 0 A	0	0.5	0	0.5	0	0	0
Kraft Roboter 1 N	0	0	0	0	0	0	0
Kraft Roboter 1 O	0	0	0	0	0	0	0
Kraft Roboter 1 A	0	-0.5	0	0	0	0	0
Moment Roboter 1 N	0	0	0	0	0	0	0
Moment Roboter 1 O	0	0	0	0	0	0	0
Moment Roboter 1 A	0	0	0	0	0	0	0
Status Greifer 0	0	0	0	0	0	0	0
Status Greifer 1	0	0	0	0	0	0	0
Entscheidungssollwert	1	1	1	1	1	1	1

Tabelle 7.4 zeigt die Korrelation der Sensoren zum Entscheidungssollwert. Da beide Greifer vor dem Ausrichten zu keinem Gegenstand Kontakt besitzen, können die Kraftwerte nicht zu Entscheidungsfindung beitragen, was sich auch in den niedrigen Korrelationswerten von 0.42 und 0.02 wiederspiegelt. Nur die Daten der Handkameras sind mit

Sensor	Korrelation
Handkameras	0.999198
Kraft Roboter 0 A	0.416322
Kraft Roboter 1 A	0.0182513

Tabelle 7.4: Korrelation der relevanten Sensordaten der Verzweigung.

den Entscheidungssollwerten korreliert (Wert: 0.999) und sind damit die zu betrachtenden Sensoren. Abb. 7.9 zeigt die Visualisierung des Projektionsvektors der berechneten ORF.

7.5 Zusammenfassung

Die in diesem Kapitel gezeigten Beispiele, der Bau des Leitwerks und des Rumpfes, demonstrieren die Funktionalität des vorgestellten Lernverfahrens. Es ist in der Lage, die Montage auch komplexere Aggregate zu erlernen und ist insbesondere fähig, aus einer Menge von internen und externen Sensoren, diejenigen zuverlässig zu ermitteln, welche für die Entscheidungsfindung bei Verzweigungen relevant sind.

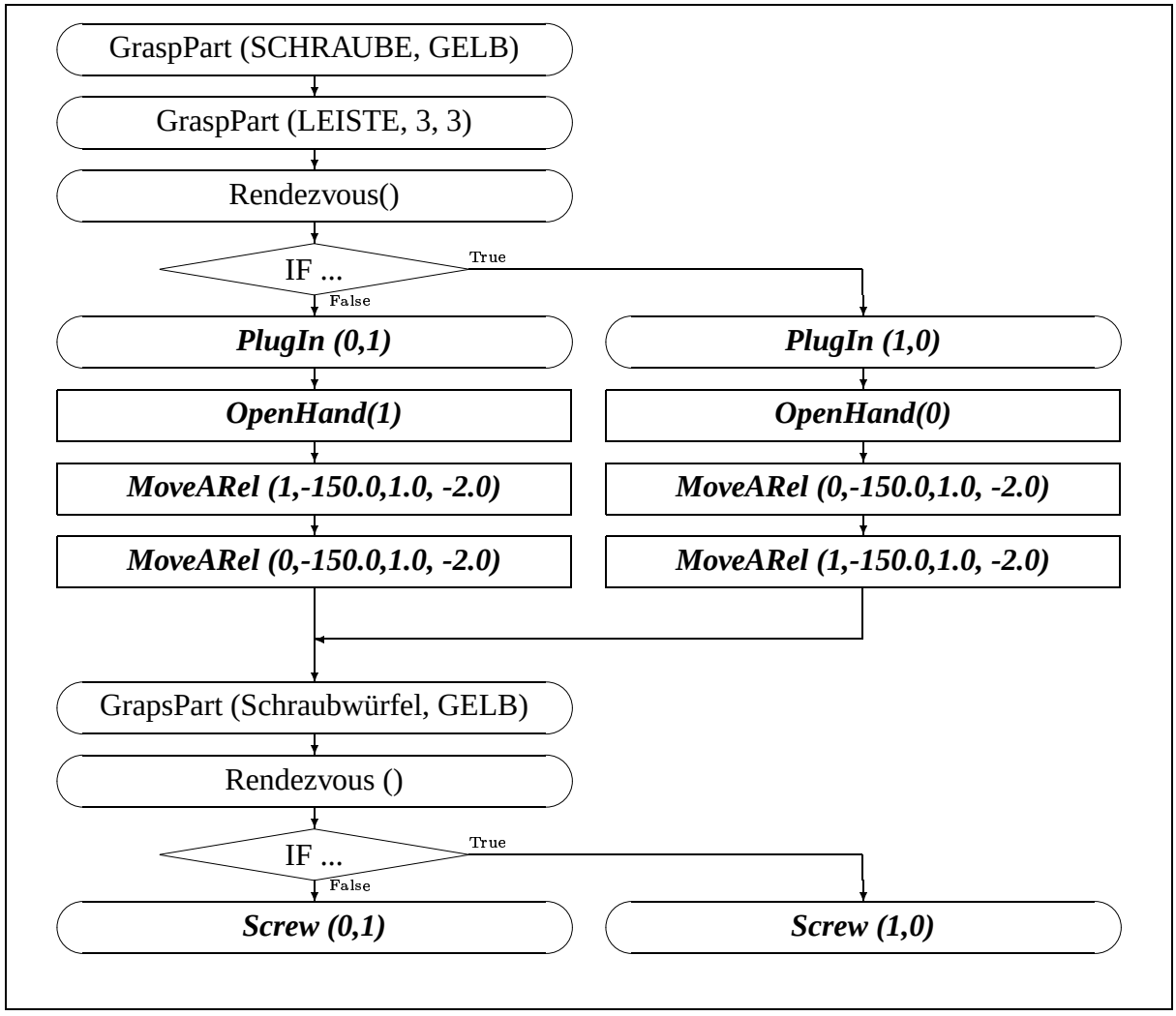


Abbildung 7.10: *Zusammengefügte Sequenz zum Bau eines Leitwerks. Die Instruktionen zur Aktualisierung der internen Repräsentation sind zur besseren Übersicht in den Flussdiagrammen nicht dargestellt.*

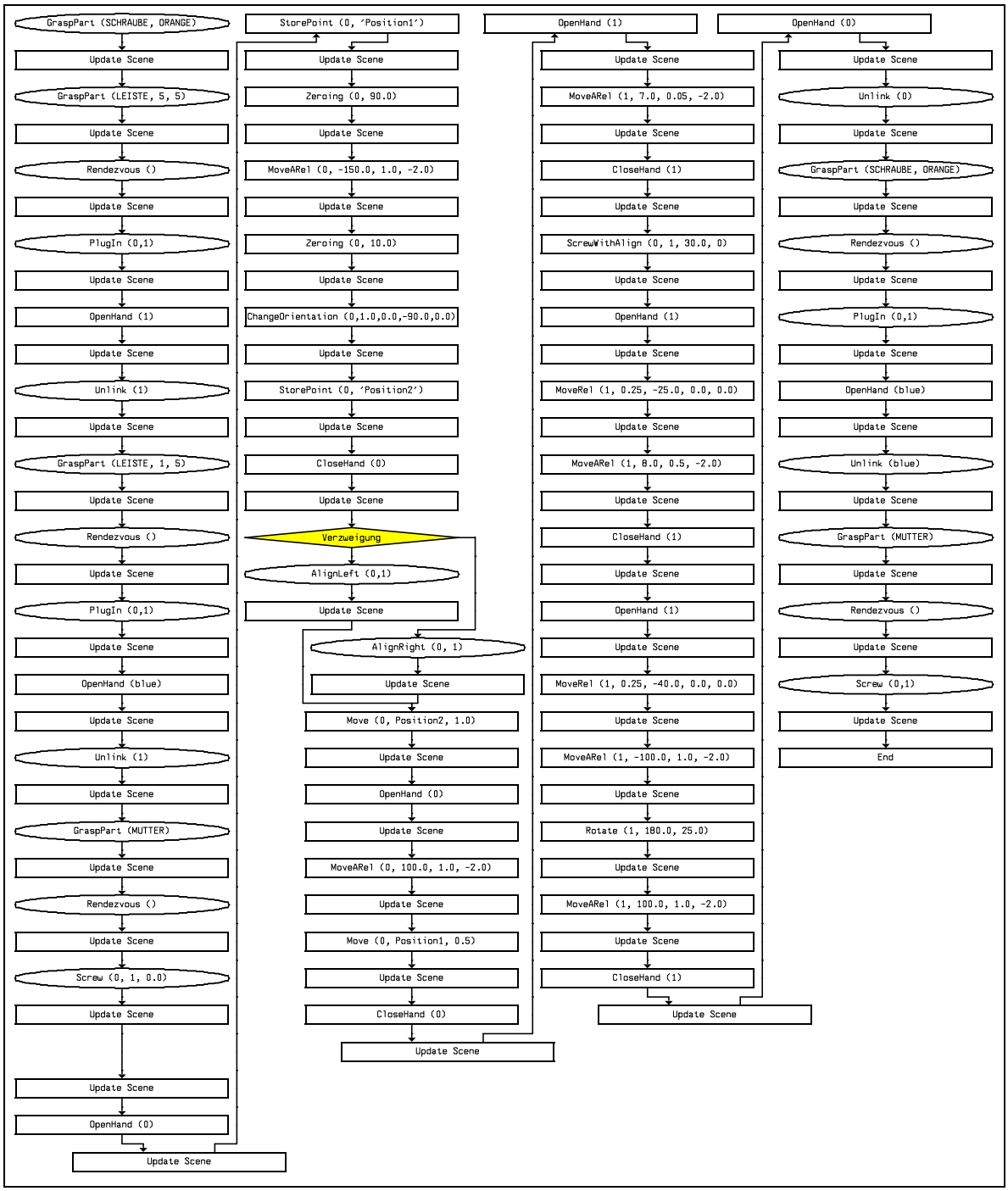


Abbildung 7.11: *Vollständiges Flussdiagramm der zusammengefügten Sequenz zum Bau eines Rumpfes.*

Kapitel 8

Zusammenfassung und Ausblick

In dieser Arbeit ist eine Kontrollarchitektur für Montageanwendungen und ein auf dieser Architektur aufbauendes Lernverfahren vorgestellt worden. Dabei wurden folgende Problemstellungen behandelt:

- Aufbau einer Entwicklungs-, Test- und Ablaufumgebung und flexiblen, deliberativen und modularen Kontrollarchitektur (*OPERA*) zur Steuerung von Montageabläufen.
- Verarbeitung von realen hochdimensionalen Sensordaten.
- Verarbeitung von Sensormustern unbekannter Struktur ohne zu Grunde legen eines Modells.
- Realisierung eines Verfahrens zur Akquirierung und Generalisierung von Montagewissen mittels Lernen.
- Montagesequenzauswahl anhand von realen Sensordaten.
- Verwendung eines realen Mehrrobotersystems mit seinen vielfältigen Unsicherheiten.

8.1 OPERA

OPERA ist eine Entwicklungs- und Ablaufumgebung, die Kernfunktionalitäten für eine Montageanwendung zur Verfügung stellt. Sie definiert eine abstrakte Schnittstelle zur Ansteuerung von Manipulatoren und Integration von Sensoren und bezieht ihre gesamte Funktionalität durch dynamisches *hinzuladen* von Modulen. Diese Module besitzen ihrerseits eine fest definierte Schnittstelle, die so leistungsfähig ist, dass alle benötigten Komponenten (Roboteransteuerung, Bewegungsprimitive, Sensoren, etc.) für ein Montagesystem über dieses Modulkonzept realisiert werden.

Zusätzlich zu der Verwaltung der Module, beinhaltet *OPERA* einen Skriptinterpreter. Dieser Interpreter führt Skripte aus, die aus Modul-, Skriptrufen und Befehlen zur Ablaufsteuerung bestehen. Mechanismen zur Fehler- und Ausnahmebehandlung während eines Montageprozesses sind ebenso vorhanden. Dadurch ist eine aufgabenorientierte Programmentwicklung mit sensorbasierten Operationen möglich.

Der Architekturansatz von *OPERA* unterscheidet sich konzeptionell nicht grundsätzlich von anderen Architekturmodellen, die einen modularen Ansatz verfolgen. Es existieren daher im Detail Unterschiede zu anderen Modellen.

Dies ist unter anderem die Schnittstelle zu den Manipulatoren, die nicht auf eine bestimmten Manipulator oder auf eine bestimmte Steuerungssoftware beschränkt und von ihrer Konzeption her sehr abstrakt ausgelegt ist. Dies hat den Vorteil, dass enge Regelschleifen sehr manipulatorbezogen implementiert werden können. Ein höherer Portierungsaufwand bei einem Wechsel auf einen anderen Manipulatortyp ist nicht zwangsläufig gegeben. Ein anderer Ansatz wäre die Definition einer Schnittstelle, die von ihren Operationen weniger abstrakt ist. In diesem Fall hätte eine allgemeine Schnittstelle für enge Regelschleifen definiert und verwendet werden müssen. Eine Schnittstelle auf dieser Abstraktionsebene ist aber sehr zeitkritisch und erzeugt hohen Verwaltungsaufwand bei der Umsetzung auf die reale Hardware. Ein deutlicher Leistungs- oder Flexibilitätsverlust des Gesamtsystems ist zu erwarten, da manche sensormotorischen Operationen nicht oder nur weniger leistungsfähig hätten realisiert werden können. Die flexible Umsetzung zwischen realen und virtuellen Roboter ist in diesem Fall nicht gegeben. Eine Schnittstelle auf hohem Abstraktionsniveau ist daher vorzuziehen.

Die konsequente Verwendung von Modulen zur Erweiterung des Systems und die definierte Schnittstelle zur Einbindung von Sensoren, ist ein weiterer Punkt, der *OPERA* von anderen Montagearchitekturen unterscheidet. Dadurch wird die Programmierung einer Systemerweiterung einfach, da nur eine Schnittstellendefinition verwendet wird. Zusätzlich ist es möglich, auch mehrere Funktionalitäten und Erweiterungen in einem Modul zu kombinieren.

Über den in *OPERA* zur Verfügung stehenden Skriptinterpreter ist es möglich Sequenzen vom Modulaufrufen zu erstellen und sowohl die Fehlerbehandlung als auch den Programmablauf zu koordinieren. Die Mechanismen sind flexibel, leistungsfähig und ermöglichen den Aufbau eines lernenden Montagesystems. Durch die Integration all dieser Komponenten ist *OPERA* nicht nur eine Entwicklungsumgebung, sondern ist auch für die Koordination des Montageablaufs zuständig, was eine Voraussetzung für schnelle Entwicklungszyklen darstellt.

Eine Vorgabe über die Architektur die Anwendung zu hierarchisieren, wie z.B. bei RCS[Alb97b], ist nicht vorhanden. Sowohl Module als auch Skripte lassen sich wechselseitig nutzen, so dass diese Komponenten keiner Komplexitätsebene zuzuweisen sind. Eine Hierarchisierung zur Komplexitätsreduzierung lässt sich aber verwirklichen und wird auch nahegelegt. Somit ist es einfacher eine deliberative Kontrollsystem mit *OPERA* zu verwirklichen als eine reaktive.

Die Integration von Entwicklung- und Ablaufumgebung, sowie das durchgängige Modulkonzept zur Ansteuerung der Roboter, Integration von Sensoren und Aufbau von

Montagefähigkeiten unterscheidet *OPERA* von den meisten anderen Kontrollarchitekturen

Das hier vorgestellte Montagesystem wird im Rahmen des Sonderforschungsbereiches 360 für die Steuerung der realen Montageabläufe erfolgreich eingesetzt und demonstriert dadurch seine Eignung für reale Anwendungen.

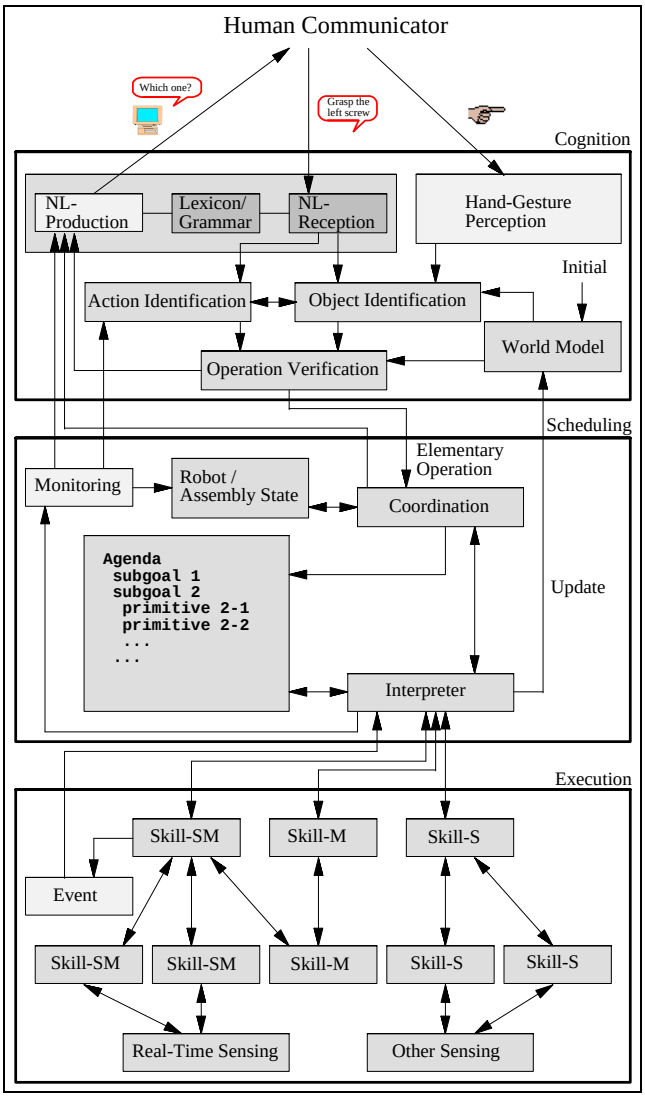


Abbildung 8.1: *Architektur des Situierten Künstlichen Kommunikators aus Sicht der Aktorik.*

Stellung von *OPERA* im SFB 360

Ein zentraler Punkt des SFB's ist auch die Umsetzung der gewonnenen Erkenntnisse in ein reales System (Demonstrator), so dass eine Herausforderung dieses Forschungs-

programms für die Robotik die Automatisierung des multisensorgestützten Montageprozesses ist.

Aus Sicht der Robotik besitzt der künstliche Kommunikator die in Abb. 8.1 gezeigte Architektur. *OPERA* deckte dabei die *Scheduling* und *Execution*-Ebene ab, während die *Cognition*-Ebene von anderen Teilprojekten realisiert wird. Die *Execution* Ebene besteht in dieser Architektur aus einfachen Montageoperationen sog. Skills, wie Schrauben, Drücken, Greifer schließen, u.s.w., die je nach Komplexitätsgrad auf anderen *Skills* aufbauen und unter Umständen direkten Zugang zu Sensordaten besitzen. Diese Ebene wird in *OPERA* durch Module abgedeckt (inkl. Roboteransteuerung) und die untereinander hierarchisiert sind.

In der *Scheduling* Ebene werden die aus der *Cognition*- Ebene empfangenen elementaren Operationen (*z.B.: "Greife eine gelbe Schraube"*) auf einzelne Montageprimitive abgebildet. Ein Montageprimitiv ist z.B. die Handlung zum Greifen eines bestimmten Objektes und wird in *OPERA* durch eine Sequenz repräsentiert, die durch den *Interpreter* ausgeführt wird. Das *Blackboard* beinhaltet den Zustand der Roboter und des Montagevorgangs.

Erweiterungen des Konzepts

Eine mögliche Erweiterung von *OPERA* ist die Option Variablenzuweisung, arithmetische Berechnungen und Programmschleifen in Skripten verwenden zu können. Dies lässt sich durch entsprechende Module realisieren, kann aber zur Verkomplizierung der Skripterstellung führen. Es ist daher die Definition einer Grammatik und Bereitstellung eines Parser vorzuziehen, um Skripte auch über eine Skriptsprache erstellen zu können. Eine zusätzliche Erweiterung ist die Möglichkeit, mehrere Instanzen von *OPERA* als ein

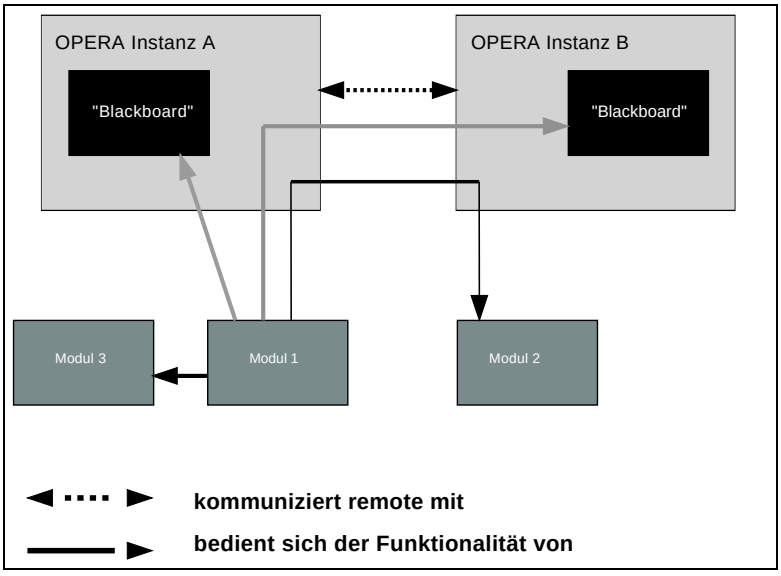


Abbildung 8.2: *OPERA als verteiltes System.*

verteiltes System in einem Rechnernetz zu betreiben (Abb. 8.2). Zur Zeit ist *OPERA* als ein monolithisches System konzipiert und es bestehen keine Mechanismen, um auf Funktionalität anderer Instanzen von *OPERA* zuzugreifen. Um dieses zu realsieren, müssten entsprechende Module implementiert werden, die die Kommunikation mit den anderen Instanzen zur Verfügung stellen. Folgende Fähigkeiten sollten vorhanden sein:

Verbindungsauf- und abbau: Verbindung zu anderen Instanzen sollten automatisch auf- und abgebaut werden. Ein Wiederaufbau einer abgebrochenen Verbindung sollte automatisch erfolgen.

Modul-Sharing: Die Nutzung von lokalen und nicht lokalen Modulen sollte transparent möglich sein, so dass sich der Benutzer nicht darum kümmern muss, bei welcher Instanz sich das aufgerufene Modul befindet. Es muss eine Art *Remote Procedure Call* eingeführt werden.

Blackboard-Sharing: Auf alle oder einen Teil der Variablen, die in einem Blackboard vorhanden sind, sollte von anderen Instanzen aus ein Zugriff möglich sein.

Um dieses zu realisieren sind aber nicht nur entsprechende Module, sondern auch Änderungen im Modulmanagement nötig, um nicht lokale Module verwalten zu können. Darüberhinaus werden Mechanismen benötigt, um den Zugriff auf nicht lokale Blackboard-Variablen zu koordinieren.

Ein weiterer Punkt ist die engere oder erweiterte Schnittstellendefinition von Modulen. Zur Zeit ist die Wahl und die Reihenfolge der Parameter eines Moduls beliebig und wird vom Programmierer festgelegt. Dies ermöglicht es dem in dieser Arbeit vorgestellten Lernverfahren nicht, weitere Annahmen über die betrachteten Instruktionen zu machen. So ist es z.B. ohne weitere Kenntnisse des Modul nicht möglich herauszufinden, ob und welcher Roboter als Parameter einem Modul übergegeben wurde. Weitere Informationen über die Art und die Reihenfolge der Instruktionen ist aber für eine bessere Beurteilung von Unterschieden in Teilsequenzen wünschenswert. Ein Lösungsansatz ist die Definition weiterer Methoden über die ein Modul weitere Informationen über seine Semantik und die Semantik seiner Parameter zur Verfügung stellt.

8.2 Lernen und Generalisieren von Montagesequenzen

Das Montagesystem lernt die Montage von (Teil-)Aggregaten, indem der Instrukteur dem System Schritt für Schritt über sprachliche Anweisungen die durchzuführende Handlung angibt. Die entsprechenden Instruktionen werden zusammen mit den vor der Ausführung gültigen Sensordaten als ein Skript in *OPERA* abgespeichert. Zusätzlich wird zur jeder Instruktion eine weitere Instruktion generiert, die eine interne symbolische Repräsentation der Umwelt mit der Realität abgleicht. Die durch den Lernvorgang entstandene Sequenz ist ein Beispiel für den Bau eines gewünschten Aggregates. Je mehr unterschiedliche Szenerien für den Bau eines Aggregates existieren, um so mehr

Beispielsequenzen werden benötigt, um in alle Situationen die Montage des Aggregates durchführen zu können.

Die zur Verfügung stehenden Beispielsequenzen werden zu einer Sequenz fusioniert, wodurch aufgrund unterschiedlicher Handlungsstränge in den einzelnen Beispielen Verzweigungen aufgebaut werden müssen. Anhand der in den Beispielsequenzen gespeicherten Sensordaten, werden die für eine Entscheidung relevanten Daten extrahiert und über einen B-Spline Fuzzy Regler für eine Entscheidungsfindung fusioniert.

Das Endprodukt ist eine Montageskript für ein (Teil-)Aggregat, durch die das Montagesystem in der Lage ist, dieses auch in unterschiedlichen Situationen autonom zusammenzubauen. Zusätzlich ist das Montagesystem fähig, durch Modifizieren der Montageskriptes das erlernte Aggregat mit anderen als den erlernten Bauteilen zu montieren.

Da die Arbeit im Rahmen des Sonderforschungsbereiches 360 entstanden ist, ist das Konzept, das System über sprachliche Anweisungen und Gesten zu instruieren, vorgegeben. Im Gegensatz zu verbreiteten Verfahren ist dem System zu Beginn der Montage sowohl das Zielaggregat als auch durchführbare Montagesequenzen nicht bekannt. Eine symbolische Repräsentation der Montagesequenzen und des Zielaggregates kann erst während der Montage aufgebaut werden. Somit wird ein gänzlich verschiedener Ansatz als z.B. in [Mos00] vorgestellt, um einen Bauplan aufzustellen. Im Unterschied zu Ansätzen des *Programming bei Demonstration* (siehe Kapitel 2.3) wird die Montage eines komplexen Aggregates und nicht einer einzelnen Montageoperation erlernt. Es wird von bestimmten Grundfertigkeiten des Systems ausgegangen. Um diese zu erlangen, ist der in [Kai96] vorgestellt Ansatz denkbar.

Die Repräsentation des Montagevorganges als ein Skript hat sich als ein angemessenes Mittel erwiesen. Die Abspeicherung einer Instrukteursanweisung als zwei Instruktionen ist ein technisches Detail und kann auch auf andere Weise gelöst werden. So ist es denkbar, eine Instruktion der SI um eine optionale Meta-Instruktion zu erweitern. Die Mitführung einer internen symbolischen Repräsentation ist essentiell für spätere Verarbeitungsschritte. Durch diese Repräsentation ist das System erst in der Lage, Handlung und Ursache, wenn auch nur eingeschränkt, zu erkennen und Montagefolgen simulativ auszuführen und damit auszuprobieren. Die Modifikation von Montagesequenzen ist erst möglich, wenn auch das Wissen über Semantik von Instruktionen vorhanden ist. Soweit bekannt existieren keine Arbeiten über die Modifikation von Montagegraphen aufgrund von Bauteiländerungen. Diese Repräsentation von Montageabläufen besitzt gegenüber den in Kapitel 2.2 vorgestellten, eine größere Flexibilität. Wie in Kapitel 2.2.1 erläutert, werden dort meist bestehende Verbindungen nicht mehr gelöst und Bauteile ändern nach einer Montageoperation nicht mehr ihre relative Position zueinander. Der Bau des Leitwerk oder des Rumpfes ist über eine solche Repräsentation nicht mehr abgedeckt, da das Aufstecken einer Leiste auf eine Schraube nicht als gültige Montageoperation gilt. Die Leiste ist nicht arretiert und kann dadurch bei Folgeoperationen ihre Lage und Orientierung spontan ändern. Es besteht keine feste Verbindung. Ein weiterer großer Vorteil des hier vorgestellten Ansatzes ist die Möglichkeit des Montagesystems aufgrund von Sensordaten während der Montage eine entsprechende Montagevariante

auszuwählen. Dies ist bei einer Montagesequenz, die aus einem Montagegraph extrahiert wurde nicht möglich, da zum Zeitpunkt der Sequenzgenerierung keine Sensordaten zur Analyse zur Verfügung stehen. Das in dieser Arbeit vorgestellte Verfahren besitzt daher eine deutlich höhere Flexibilität in der Auswahl der zu bauenden Aggregate.

Das in Kapitel 4.4 vorgestellte Fusionsverfahren hat sich in verschiedenen Anwendungen als zuverlässig erwiesen. Das Verfahren ist im Gegensatz zu vielen anderen Verfahren nicht modellbasiert und ist trotzdem in der Lage verschiedene Sensordaten miteinander zu fusionieren. Es ist einfach in der Anwendung und Implementierung. Es benötigt abgesehen vom Training wenig Rechenzeit. Da das Lernen auf einem statistischen Ansatz beruht, ist bei der benötigten Anzahl von Beispielmontage zu berücksichtigen.

Erweiterungen des Konzepts

Weitergehende Arbeiten sind weiterhin im Bereich der Akquirierung von Beispielsequenzen notwendig. Wie im Kapitel 5 erörtert, ist es bei Fehlerfällen problematisch, die Montage weiterzuführen, da das Erreichen eines früheren Montagezustandes nicht trivial ist. Mögliche Lösungen sind in Kapitel 5.2 aufgezeigt. Eine weitere Möglichkeit ist, die Fehlerbehandlung mit in den Lernprozess aufzunehmen und die entsprechende Fehlerbehebung beim Wiederauftreten der Ausnahme wiederzuverwenden.

Werden eine ganze Reihe von Beispielmontagen dem System gezeigt, so wird die Anzahl der zu speichernden Daten immer größer. Die Sensordaten werden nicht nur während des Lernprozesses, sondern auch bei der späteren Wiederholung der gelernten Montagesequenzen aufgezeichnet. Da sich aber schon nach den ersten Montagebeispielen abzeichnet, welche Sensordaten bei einer Verzweigung in der Sequenz relevant sind oder nicht, ist es möglich, die nicht benötigten Sensordaten gar nicht erst aufzuzeichnen und damit Ressourcen zu sparen.

Muss während der Ausführung einer Sequenz die boolsche Funktion einer Verzweigung berechnet werden, so verwendet das in Kapitel 4.4 vorgestellte Verfahren nur zu diesem Zeitpunkt gültige Sensordaten. Sensordaten vor diesem Zeitpunkt werden nicht betrachtet. Eine mögliche Erweiterung ist die Einbeziehung dieser Sensordaten über neuronale Netze vom Typ ART *(Adaptive Resonance Theory)* [CG87, WDMA96, AdA98]. Diese Netzarchitekturen haben die Eigenschaft, dass die Plastizität für neue Muster beibehalten, aber gleichzeitig zu starke Modifikationen bereits gelernter Muster verhindern. In [WDMA96] werden mit einem neuronalen Netz des Typs ART2 die unterschiedlichen Zustände eines Reaktors beurteilt und klassifiziert. Ein anderer Ansatz wird in [SRC01] vorgestellt. Dort werden von einem Prozess generierte Zeitreihen mit Hilfe eines modellbasierten Bayesian Algorithmus gruppiert (*clustering*). Das Verfahren wird angewendet, um z.B. den Verlauf von Daten mehrerer Sensoren eines mobilen Roboters einem bestimmten Prozess oder einer bestimmten Ereignis zuzuordnen. Dieses Verfahren könnte genutzt werden, um nicht nur aus der aktuellen Situation heraus zu entscheiden, sondern Unterschiede in den Handlungsabläufen im Zusammenhang

unterscheiden zu können.

Ist die interne Repräsentation der Montage hinreichen genau, so kann die zum Aufbau eines Montagegraphen verwendet werden (siehe Kapitel 2.2), wodurch weitere Fähigkeiten durch die Anwendung der klassischen Verfahren auf diese Graphen zur Verfügung gestellt werden können.

Anhang A

Technische Aspekte von *OPERA*

A.1 Benutzeroberfläche von *OPERA*

Komponenten

Nach dem Starten von *OPERA* über das Kommando `opera`, werden zwei Fenster geöffnet, das Hauptfenster (Abb A.1) und das Sensorfenster (Abb. A.2).

Hauptfenster

Das Hauptfenster besteht aus folgenden Einheiten:

Hauptmenü: Über das Hauptmenü lassen sich alle Standardfunktionen von *OPERA* aufrufen. Dies umfasst das Öffnen von Sequenzen, das Nachladen von Modulen sowie das direkte Ausführen von Modulen.

Erweitertes Menü: Im erweiterten Menü können Module ihrerseits Menüpunkte platzieren und damit die Oberfläche von *OPERA* erweitern.

Roboter Status: In diesem Fenster wird der Status der zur Verfügung stehenden Roboter angezeigt. Die Informationen umfassen die Namen der verfügbaren Roboter und welche Roboter von *OPERA* belegt werden.

Aktuelle Sequenz: In diesem Fenster wird die Sequenz angezeigt, die gerade vom Sequenzinterpreter (siehe Kapitel 3.3.2) ausgeführt wird. Die Instruktion, die gerade interpretiert wird, ist durch einen Balken markiert.

Meldungsfenster: In diesem Fenster werden Meldungen aller Art ausgegeben.

Aktuelle Bildanzeige: In diesem Fenster können Module Bilder darstellen.

Nach dem Öffnen der Fenster startet das Montagesystem und führt eine Reihe von Initialisierungen durch. Unter anderem wird die Konfigurationsdatei von *OPERA* verarbeitet (siehe Kapitel A.1). Erst wenn die Meldung

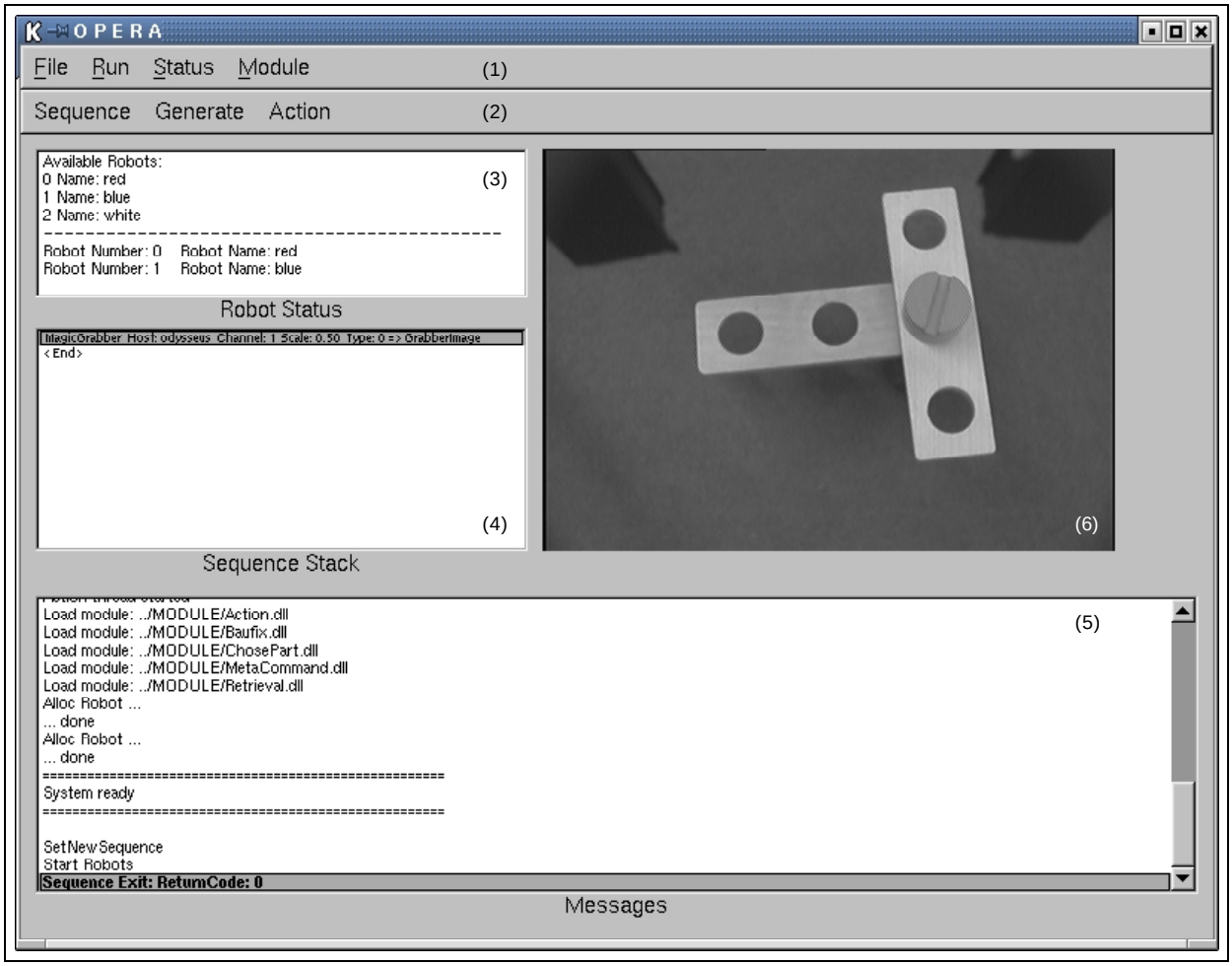


Abbildung A.1: *Hauptfenster von* OPERA *(Hauptmenü (1), Erweitertes Menü (2), Roboterstatus (3), aktuelle Sequenz (4), Meldungsfenster (5), aktuelle Bildanzeige (6)).*

```
==================================================
System ready
==================================================
```

im Meldungsfenster erscheint, ist die Initialisierung abgeschlossen und es kann mit dem System gearbeitet werden[1].

Sensorfenster

Im Sensorfenster werden die Informationen von Sensormodulen (siehe Kapitel 3.3.4) visualisiert. Jedes Modul, welches der Kategorie `MODULE_SENSOR` angehört, hat in diesem Fenster die Möglichkeit seine Sensordaten darzustellen. Das Fenster ist als ein

[1]Da das X-Windows System mancher UNIX-System nicht Multithread-safe ist, sollten während der Initialisierung keine Menüfunktionen aufgerufen werden.

Karteikartensystem organisiert und die Daten der einzelnen Module werden über *Reiter* selektiert. Die Abb. A.2 zeigt die Kraftmomentendaten von zwei Robotern, während

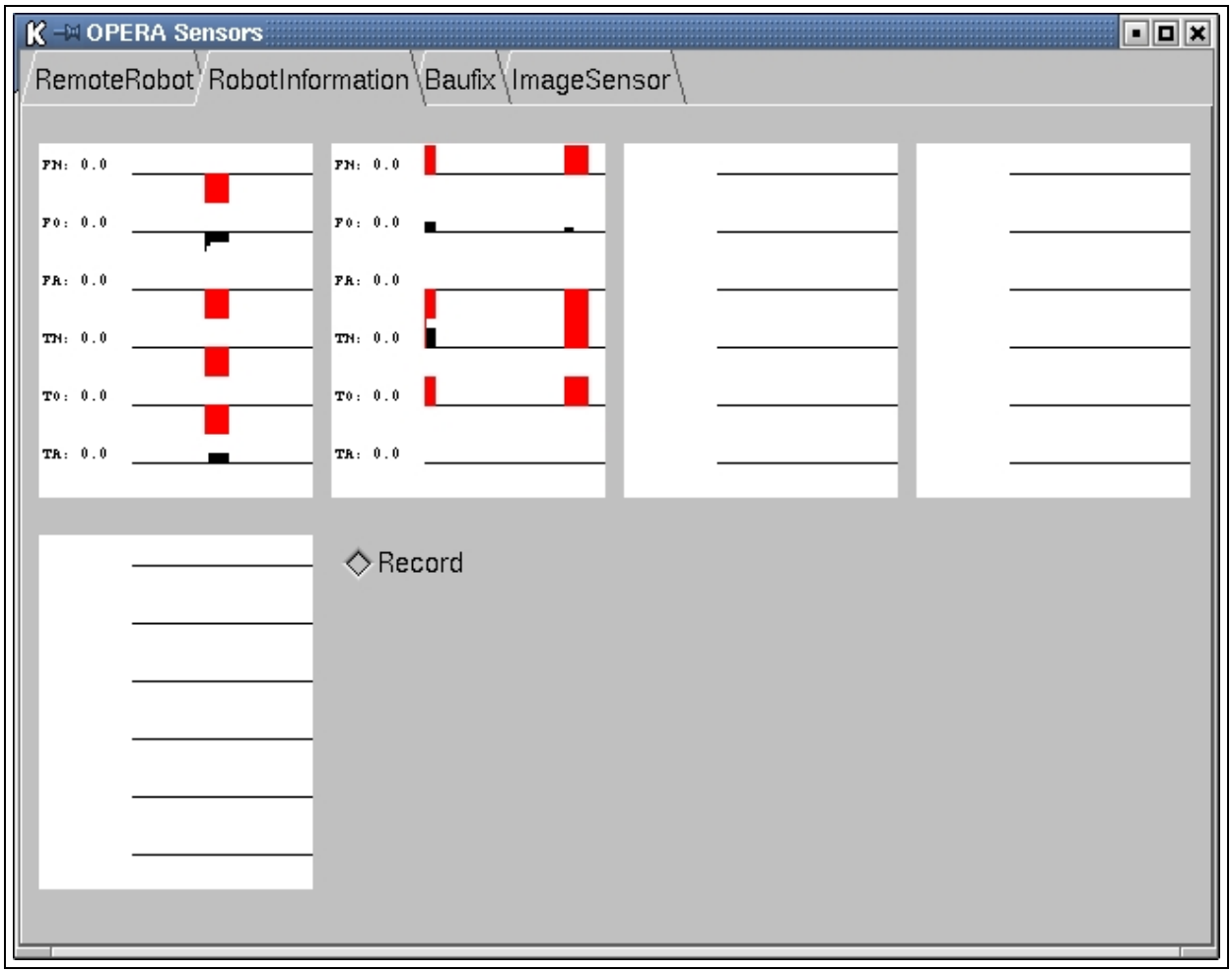


Abbildung A.2: *Sensorfenster von* OPERA. *Die Abbildung zeigt die visualisierten Daten von zwei Kraftmomentensensoren, deren Werte über die Zeit aufgetragen sind.*

die Positionsinformationen vom Roboter und der Bildsensoren verdeckt sind. Die Bedienung der einzelnen Karteikarten ist abhängig von der Implementation des jeweiligen Moduls.

Menüelemente

Beschreibung des Menüs File

Über diesen Menüpunkt werden neue oder vorhandene Sequenzen geöffnet, Module zum System hinzugeladen und die Anwendung beendet.

New Sequence erzeugt eine neue Sequenz. Als erstes müssen die Parameternamen der Sequenz angegeben werden (Abb. A.3). Die Namen werden durch Kommas

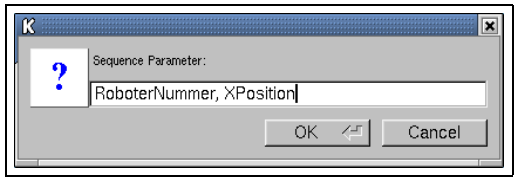


Abbildung A.3: *Dialog für die Sequenzparameter.*

getrennt. Die Typen der Parameter ergeben sich erst zur Laufzeit. Nach Angabe der Namen öffnet sich ein weiteres Fenster über das die Sequenz erstellt wird (siehe Kapitel A.1). Die Sequenzparameter können dabei als Parameter für Modul- und Sequenzaufrufe verwendet werden.

Open Sequence öffnet eine vorhandene Sequenz. Es erscheint ein Dialogfenster über das man die entsprechende Datei[2] öffnet (Abb. A.4).

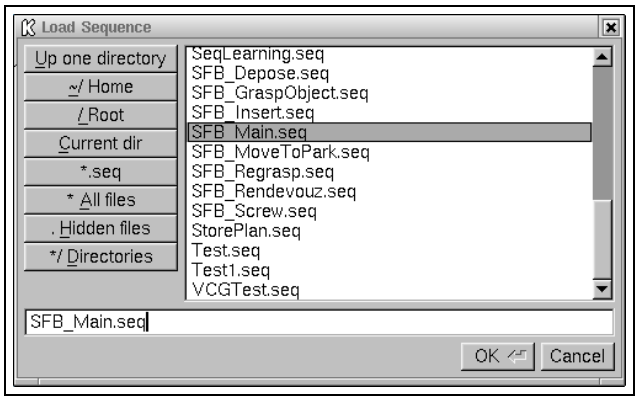


Abbildung A.4: *Dialog zur Dateiauswahl.*

Open Modul: Über diesen Menüpunkt kann ein Modul zum System hinzugeladen werden. Es erscheint ein Dialogfenster über das die entsprechende Datei[3] ausgewählt wird. Wenn das Modul geladen werden konnte, so steht es sofort in allen entsprechenden Menüs zur Verfügung. Konnte es nicht geladen werden, erscheint eine Fehlermeldung im Meldungsfenster.

About öffnet einen Dialog mit Versions- und Plattforminformationen.

Exit beendet *OPERA*. Es werden alle Module aus dem System entfernt und die Anwendung geschlossen.

[2] Dateien in denen Sequenzen abgespeichert sind haben üblicherweise die Endung `.seq`.
[3] Module haben üblicherweise die Endung `.dll`.

Beschreibung des Menüs Run

Über dieses Menü werden einzelne Roboter reserviert, freigegeben und die aktuelle Ausführung unterbrochen.

Alloc Robot reserviert und belegt einen Roboter. Es erscheint ein Dialog, über den ein weiterer Manipulator reserviert und belegt wird (Abb. A.5). Die Roboter werden

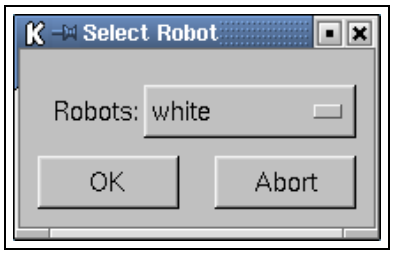


Abbildung A.5: *Dialog zur Roboterauswahl.*

in der Reihenfolge ihrer Reservierung mit Null beginnend durchnummeriert. Unter dieser Nummer können sie dann aus einem Modul oder einer Sequenz angesprochen werden.

Dealloc Robot hebt die Belegung aller Roboter wieder auf.

Stop stoppt die aktuelle Ausführung einer Sequenz oder eines Moduls.

Beschreibung des Menüs Status

Über dieses Menü können Statusinformationen abgerufen werden.

Module listet alle geladenen Module auf (Abb. A.6). Pro Zeile werden folgende Informationen zu einem Modul aufgelistet:

- Dateiname des Moduls;
- wie oft wird das Modul von einer Sequenz verwendet;
- ist das Modul automatisch oder manuell geladen worden;
- Versionsnummer;
- Kategorien, der das Modul angehört (r: Run, I: Interactive, E: Event, S: Sensor, R: Robot) (siehe Kapitel A.2).

Event-Modules zeigt alle Module an, die der Kategorie `MODULE_EVENT` angehören (siehe Kapitel 3.3.3).

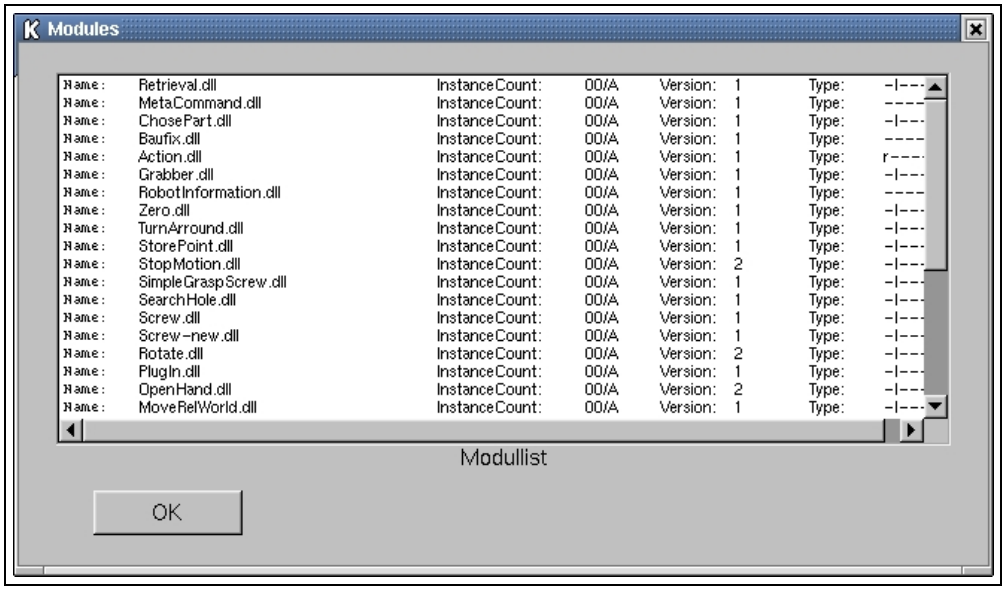


Abbildung A.6: *Liste aller geladenen Module.*

Beschreibung des Menüs **Module**

Über dieses Menü lassen sich einzelne Module der Kategorie `MODULE_INTERACTIVE` direkt ausführen. Nach der Anwahl eines Moduls wird die Angabe der Aufrufparameter erwartet. Diese werden durch einen Textstring angegeben, bei dem die Parameter durch Kommas getrennt werden. Der Dialog gibt eine kurze Hilfestellung, in welcher Reihenfolge welche Parameter angegeben werden müssen. Sollen die Parameter nicht über einen Text angegeben werden, so muss der Dialog über den Schalter `OK` **ohne** eine Eingabe im Textfeld verlassen werden. Es erscheint dann, soweit vorhanden, der entsprechende Dialog des Moduls, über den die Parameter angegeben werden können. Hiernach wird das Modul mit den angegebenen Parametern ausgeführt. Wird der erste oder der zweite Dialog über den mit `Cancel` beschrifteten Schalter verlassen, wird die Aktion abgebrochen.

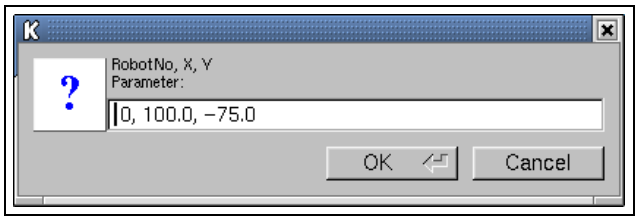


Abbildung A.7: *Parameterangabe über einen Textstring.*

Bei Eingabe der Parameter als Text müssen folgende Punkt beachtet werden:

- Parameter werden durch Kommas getrennt.

- Texte müssen mit Hochkommas gekennzeichnet werden.
- Zahlen ohne einen Dezimalpunkt werden als Fixkommazahlen betrachtet.
- Zahlen mit Dezimalpunkt werden als Fließkommazahlen interpretiert.
- Ist ein Parameter weder ein Text, eine Fix- oder Fließkommazahl, so wird er als Name für eine Variable im *Blackboard* interpretiert. Ist eine Variable unter diesem Namen im *Blackboard* vorhanden, so wird der Inhalt als Parameter an das Modul weitergereicht.

Abb. A.7 zeigt die Angabe von Parametern über einen Textstring. Es wird die Nummer des Roboters als Fixkommazahl und die eine Position in der Ebene über Fließkommazahlen angegeben.

Erstellen einer Sequenz

Sequenzen werden in einem Sequenzfenster bearbeitet (Abb. A.8). In diesem Fenster befindet sich ebenfalls ein Menü mit den notwendigen Funktionen zur Bearbeitung der Sequenz. Über die Auswahl der gewünschten Kommandos und Kontrollstrukturen wird die Sequenz zusammengesetzt. Die neuen Instruktionen werden entweder am Ende oder an der selektierten Stelle eingefügt.

Menüstruktur des Sequenzfensters

File: **Save** sichert die Sequenz unter ihrem Namen.

Save As sichert die Sequenz unter einem anderen Namen. Es erscheint ein Dialog mit dessen Hilfe der neue Namen angegeben wird.

Close schießt das Fenster mit der Sequenz. Ist die Sequenz verändert, aber nicht gesichert worden, so erfolgt eine Sicherheitsabfrage, ob die Sequenz gesichert werden soll.

Edit: **Edit Parameter:** Über diesen Punkt können die Parameternamen der Sequenz geändert werden.

Edit öffnet den entsprechenden Dialog zum Ändern der Instruktionsparameter.

Copy kopiert die selektierte Instruktion ins Clipboard.

Cut verschiebt die selektierte Instruktion ins Clipboard.

Paste kopiert die Instruktion aus dem Clipboard an die selektierte Position in der Sequenz. Die selektierte Instruktion verschiebt sich um eine Position nach hinten.

Control: **Label** fügt eine Markierung in die Sequenz ein. Über einen Dialog muss ein eindeutiger Label-Name angegeben werden.

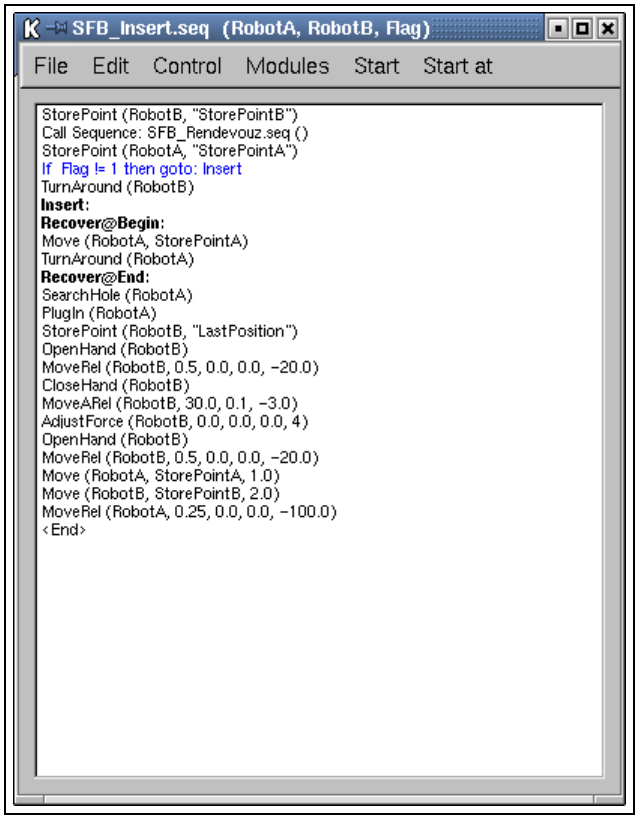


Abbildung A.8: *Sequenzfenster mit Beispielsequenz.*

Condition fügt eine IF-THEN Instruktion in die Sequenz ein. Über einen Dialog (Abb. A.9) muss die Bedingung angegeben werden, wobei die Syntax der Skriptsprache TCL zu verwenden ist. Verwendet werden können neben Texten, Fix- und Fließkommazahlen auch Variablen aus dem *Blackboard*. Ebenfalls muss eine Markierung angegeben werden, die angesprungen wird, wenn die Bedingung erfüllt ist.

Jump fügt einen Sprungbefehl in die Sequenz ein. Es ist eine Markierung anzugeben, bei der die Ausführung des Skriptes fortgeführt werden soll.

Exec Sequence fügt einen Sequenzaufruf in die Sequenz ein. Die Parameter sind als Textstring anzugeben.

RecoverPoint fügt den Anfangs- und Endbefehl einer Recover- Umgebung ein (siehe Kapitel 3.3.3).

CatchPoint fügt den Anfangs- und Endbefehl einer Catch- Umgebung ein (siehe Kapitel 3.3.3).

Modules fügt das ausgewählte Modul in die Sequenz ein. Die Parameterangabe erfolgt analog zum Modul-Menü im Hauptfenster.

Start startet die Ausführung der Sequenz.

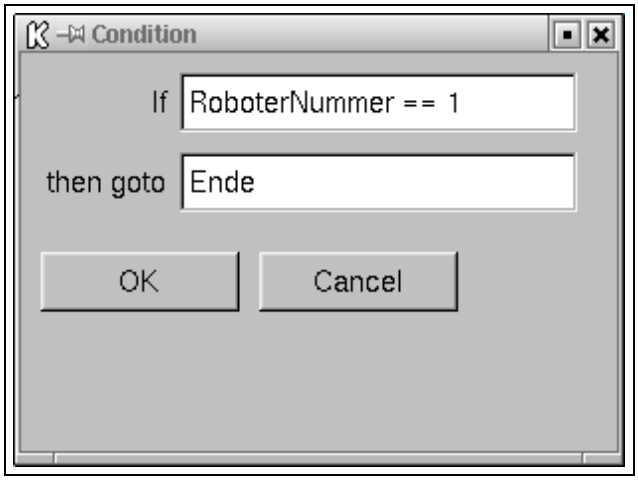


Abbildung A.9: *Dialog zum Erstellen einer* IF-THEN *Instruktion.*

Start At startet die Ausführung der Sequenz ab der selektierten Instruktion.

Konfigurationsdatei

Bei Start von *OPERA* wird eine Datei `.operarc` abgearbeitet. Sie wird im aktuellen Verzeichnis gesucht, muss aber nicht vorhanden sein. In ihr stehen die beim Start von *OPERA* zu ladenden Module; pro Zeile eine Modul. Die Einträge können zum Beispiel so aussehen:

```
../MODULE/Robot.dll
../MODULE/AssemblyEvent.dll
../MODULE/CloseHand.dll
../MODULE/InitSystem.dll
../MODULE/Move.dll
../MODULE/MoveJ.dll
../MODULE/OpenHand.dll
../MODULE/StopMotion.dll
#
../MODULE/RobotInformation.dll
```

Zeilen, deren erstes Zeichen eine # ist werden ignoriert. Zusätzlich können auch die Roboter beim Programmstart automatisch belegt werden. Die entsprechenden Einträge lauten z.B.:

```
Alloc: 0
Alloc: 1
```

Es wird zuerst der erste verfügbare Roboter belegt und dann der zweite (die Zählung beginnt bei Null). Diese Einträge dürfen nur gemacht werden, wenn auch Roboter zur Verfügung stehen: in diesem Beispiel das Modul `Robot.dll`.

A.2 Programmierung eines *OPERA*-Moduls

Ein *OPERA*-Modul ist aus Sicht des Betriebssystem eine *Shared Library*, die zur Programmlaufzeit nachträglich *gelinkt* wird. Damit sie vom Gesamtsystem als Modul erkannt werden kann, muss diese Library mehrere Bedingungen erfüllen:

- Es muss eine Funktion **`CreateModuleInstance`** definiert sein.
- Die Funktion **`CreateModuleInstance`** muss bei Aufruf eine Instanz einer Klasse erzeugen, die von der Klasse **`TRunModule`** abgeleitet ist.
- Die Instanz der Klasse, die von **`CreateModleInstance`** erzeugt worden ist, muss die Methoden **`GetCommandName, GetType, operator()`** überladen.

Nachfolgend wird detailierter auf diese Funktionen und Methoden eingegangen[4].

Standardmethoden

`CreateModuleInstance` Diese Funktion ist in einer *Shared Library* zu definieren, soll diese von *OPERA* als Modul erkannt werden. Diese Funktion wird nur einmal nach dem dynamischen *linken* des Moduls aufgerufen und erzeugt eine Instanz des in dem Modul definierten Modulobjektes.

```
1 class TOwnModuleClass;
2
3 extern "C" TRunModule* CreateModuleInstance (void);
4
5 TRunModule* CreateModuleInstance (void) {
6   return new MyOwnModuleClass;
7 }
```

`TRunModule::GetCommandName` Diese Methode von der Klasse **`TRunModule`** ist zu überladen und liefert den Namen unter dem das Modul im Gesamtsystem zu identifizieren ist. Zwei Module dürfen nicht ein und denselben Namen besitzen, da sie sonst nicht unterschieden werden können. Üblicherweise bezeichnet der Name die vom Modul definierte Operation.

```
1 class MyOwnModuleClass;
2
3 char* MyOwnModuleClass::GetCommandName (void) {
4   return "Show_Number";
5 }
```

[4] *OPERA* verwendet für die Benutzerschnittstelle die Funktionsbibliothek FLTK [DOC]. Daher sollte die grundlegende Arbeitsweise von FLTK zum besseren Verständnis bekannt sein.

`TRunModule::GetType` Diese Methode liefert den Typ des Moduls zurück und gibt an, zur welcher Kategorie (*INTERACTIV, SENSOR, RUN, ...*) es gehört. Je nachdem zu welchem Typ das Module gehört, wird die Methode **`operator()`** zu unterschiedlichen Zeitpunkten mit unterschiedlichen Parametern aufgerufen, wobei ein Modul gleichzeitig mehreren Kategorien angehören kann. Die entsprechenden Makrodefinitionen müssen in diesem Fall mit ODER verknüpft werden.

```
1 class MyOwnModuleClass;
2
3 int MyOwnModuleClass::GetType (void) {
4   return MODULE_INTERAKTIVE | MODULE_RUN;
5 }
```

Kategorie `MODULE_RUN`: Gehört ein Modul zu dieser Kategorie, so wird die Methode **`operator()`** sofort nach dem Laden des Moduls aufgerufen. Das Modul hat dann die Möglichkeit, weitere Initialisierungen durchzuführen, die im Konstruktor noch nicht gemacht werden konnten.

Kategorie `MODULE_INTERACTIVE`: Diese Kategorie bezeichnet ein Modul, bei dem die Parameter für die Methode **`operator()`** interaktiv geändert werden können. Soll ein Modul in eine Sequenz aufgenommen werden, so muss es dieser Kategorie angehören. Da in *OPERA* auf zwei unterschiedliche Arten die Parameter für ein Modul angegeben werden können, ist es nicht unbedingt notwendig, für diesen Fall die Methode **`Edit`** ebenfalls zu überladen. Es ist aber empfehlenswert.

Kategorie `MODULE_EVENT`: Ein Modul, das dieser Kategorie angehört, wird aufgerufen, wenn bei Ablauf einer Sequenz die Ausnahme **`TOPERAEvent`** geworfen wird. Das System wird in diesem Fall alle Module, die dieser Kategorie angehören, nacheinander aufrufen, bis ein Modul durch seinen Rückgabewert signalisiert, dass es diese Ausnahme behandelt hat.

Kategorie `MODULE_SENSOR`: Ein Modul dieser Kategorie steuert einen Sensor an und zeigt diese Daten im *Sensorfenster* an (siehe Kapitel A.1).

`TRunModule::operator()` Diese Methode der Klasse **`TRunModule`** definiert die eigentliche Funktionalität eines Moduls. Das Aufrufen eines Moduls entspricht immer dem Aufruf der Methode **`operator()`** und es hängt vom Typ des Parameters ab, welche Funktionalität das Modul zu leisten hat. Der Parameter dieser Methode ist ein Zeiger auf **`TOCCLObject`** und ist damit sehr allgemein gehalten.

```
1 class MyOwnModuleClass;
2
3 int MyOwnModuleClass::operator() (TOCCLObject *Arg) {
```

Gehört das Modul der Kategorie **`MODULE_RUN`** an, so wird diese Methode mit einem Null-Zeiger aufgerufen.

```
1  if (Arg == NULL) {
2    // Do something
3    return MODULE_RETURNCODE_NOERROR;
4  }
```

Bei einem Modul der Kategorie **MODULE_INTERACTIVE** ist der Typ des Parameters nicht vorgeben. Er kann vom Typ **TParameterList*** oder eines Typs den der Programmierer der Methode **TRunModule::Edit** gewählt hat, sein.

```
1  TParamaterList *Param;
2  TOCCLLong      Value;
3
4  if (dynamic_cast<TParameterList*>(Arg) != NULL) {
5    Param = dynamic_cast<TParameterList*>(Arg);
6
7    try {
8      Value = Param->GetLongParam (1);
9    }
10   catch (TOPERAEvent &Error) {
11     ShowMessage ("Wrong Parameter");
12     return MODULE_RETURNCODE_UNDEFERROR;
13   }
14   // do something
15
16   return MODULE_RETURNCODE_NOERROR;
17 }
18
19 if (dynamic_cast<MyOwnParamClass*>(Arg) != NULL) {
20   // Do something
21   return MODULE_RETURNCODE_NOERROR;
22 }
```

Ein Modul, das der Kategorie **MODULE_EVENT** angehört, wird beim Auftreten einer *Exception* mit dem Type **TOPERAEvent** mit der Instanz dieser Klasse aufgerufen.

```
1  TOPERAEvent *Error;
2
3  if (dynamic_cast<TOPERAEvent*>(Arg) != NULL) {
4    Error = dynamic_cast<TOPERAEvent*>(Arg);
5    // Handle exception
6
7    return EVENTRETURNCODE_CONTINUE;
8  }
```

Für diesen Fall des Aufrufs existieren mehrere Rückgabewerte:

EVENTRETURNCODE_NOTHANDLED wird zurückgeliefert, wenn das Modul für diese Ausnahme nicht zuständig war und sie somit nicht bearbeitet hat. Das System sucht dann nach einem anderen Modul, das die Ausnahme behandelt.

EVENTRETURNCODE_CONTINUE gibt an, dass der Ablauf der Sequenz weiter fortfahren kann.

EVENTRETURNCODE_ONCEMORE wird zurückgeliefert, wenn die Ausführung des Moduls, welches die Ausnahme verursacht hat, wiederholt werden soll.

EVENTRETURNCODE_RECOVER wird in dem Fall von der Methode zurückgegeben, wenn der Ablauf der Sequenz ab einem definierten Aufsetzpunkt wiederholt werden soll.

EVENTRETURNCODE_ERROR als Rückgabewert bewirkt den weiteren Ablauf der Sequenz ab einer definierten Stelle.

EVENTRETURNCODE_ABORT bricht die Ausführung insgesamt ab.

Ist das Modul ein Sensor-Modul (Kategorie **MODULE_SENSOR**) so wird es nach dem Laden mit dem Parameter vom Typ **TSensorTabDescription*** aufgerufen.

```
1 struct TSensorTabDescription : public TOCCLObject {
2   TSensorWindow *MainWindow;
3   Fl_Group      *Group;
4 };
```

In dieser Struktur sind die notwendigen Informationen, um im Sensor- Fenster die notwendigen Elemente zur Visualisierung der Sensordaten zu platzieren. Bei der Programmierung wird sich dabei der *Fltk-Library* [DOC] bedient und die neuen Elemente müssen der Gruppe **Group** zugeordnet werden. Dies ist der einzige Zeitpunkt, bei dem innerhalb der Methode **operator()** Funktionen verwendet werden dürfen, die auf die *X-Library* des Betriebssystems zurückgreifen.

```
1   if(dynamic_cast<TSensorTabDescription>(Arg)!=NULL){
2     Description = dynamic_cast<TSensorTabDescription*>(Arg);
3
4     Group = Description->Group;
5
6     RText = new Fl_Output(120, 50,300,30,"Red  :");
7     BText = new Fl_Output(120,100,300,30,"Blue :");
8     WText = new Fl_Output(120,150,300,30,"White:");
9
10    Group->add (RobotRedText);
11    Group->add (RobotBlueText);
12    Group->add (RobotWhiteText);
13
14    return MODULE_RETURNCODE_NOERROR;
15  }
```

Weitere Methoden

Im Folgenden werden die Methoden erklärt, die nicht unbedingt überladen werden müssen, um ein korrektes Modul für *OPERA* zu implementieren.

TRunModule::Edit Die Methode wird aufgerufen, wenn der Benutzer die Parameter für einen Modulaufruf interaktiv erstellen bzw. ändern möchte. Die Methode erhält als Parameter einen Zeiger vom Typ **TOCCLObject** und muss ebenfalls einen Zeiger vom Typ **TOCCLObject** zurückliefern.

```
class MyOwnModuleClass;

TOCCLObject* MyOwnModuleClass::Edit (TOCCLObject*);
```

Ist der Wert des Zeigers, der als Parameter übergeben wird, NULL, so ist von der Methode die notwendige Struktur für die Parameter des Modulaufrufs selbst im Speicher anzulegen und die Adresse als Rückgabewert zu verwenden. Bricht der Benutzer die interaktive Bearbeitung der Parameter ab, so ist in diesem Fall Null als Rückgabewert zu verwenden. Ist der Wert des Parameters ungleich Null, so existiert bereits eine Parameterstruktur, deren Adresse dann als Rückgabewert verwendet werden muss. Als Parameterstruktur kann ein beliebiges Objekt, welches von **`TOCCLObject`** abgeleitet ist, verwendet werden. Es ist aber die Klasse **`TParameterList`** vorzuziehen. In dieser Methode darf auf Funktionen der *X-Library* zurückgegriffen werden.

`TRunModule::GetTextFromParamString` Die Methode erhält die mit der Methode **`Edit`** erstellte Parameterstruktur als Parameter und erstellt daraus einen Text, den sie im Parameter **`Text`** ablegt.

```
class MyOwnModuleClass;

void MyOwnModuleClass::GetTextFromParamString (TOCCLObject *Arg, char *Text);
```

`TRunModule::GetParameterDescription` Entscheidet sich der Benutzer die Parameter für einen Modulaufruf nicht über die Methode **`Edit`**, sondern generisch als Zeichenkette zu erstellen, so liefert das Modul über diese Methode eine Beschreibung, in welcher Reihenfolge die Werte anzugeben sind.

```
class MyOwnModuleClass;

TOCCLString MyOwnModuleClass::GetParameterDescription(void){
  return "Number";
}
```

`TRunModule::AllocX` Will ein Modul direkt nach dem Laden Funktionen aufrufen, die zur *X-Library* des Betriebssystem gehören, so sind diese in dieser Methode durchzuführen.

`TRunModule::DeallocX` Diese Methode wird aufgerufen, bevor das Modul endgültig aus dem System entfernt wird. Eventuelle Operationen, die man in **`AllocX`** durchgeführt hat, sind in dieser Methode rückgängig zu machen.

Beispiel

Dieser Quelltext zeigt ein einfaches Modul zum Ausgeben einer Zahl auf dem Bildschirm[5].

```
// ================================================
// == INCLUDE-DATEIEN                            ==
// ================================================

#include <typeinfo>
#include "OCCL/TOCCLObject.H"
#include "OPERA/TRunModule.H"

#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Button.H>
#include <FL/Fl_Input.H>

// ================================================
// == Klassen Definition                         ==
// ================================================

class MyOwnModuleClass : public TRunModule {
public:
  MyOwnModuleClass (void);          // Konstruktor
  ~MyOwnModuleClass (void);         // Destruktor

  int operator() (TOCCLObject*);

  TOCCLObject* Edit (TOCCLObject*);

  int         GetType (void);
  void        GetTextFromParamString(TOCCLObject*,char*);
  char*       GetCommandName (void);
  TOCCLString GetParameterDescription(void);
};


// ================================================
// == C- Funktion CreateModuleInstance           ==
// ================================================


extern "C" TRunModule* CreateModuleInstance (void);

TRunModule* CreateModuleInstance (void) {
  return new MyOwnModuleClass;
}


// ================================================
// == Konstruktor und Destruktor                 ==
// ================================================

MyOwnModuleClass::MyOwnModuleClass (void) {}

MyOwnModuleClass::~MyOwnModuleClass (void) {}


// ================================================
```

[5]Weitere Beispiele finden sich auf der beiliegenden CD-ROM.

```
56 // == GetCommandName                                ==
57 // ================================================
58
59 char* MyOwnModuleClass::GetCommandName (void) {
60   return "Show_Number";
61 }
62
63
64 // ================================================
65 // == GetType                                       ==
66 // ================================================
67
68 int MyOwnModuleClass::GetType (void) {
69   return MODULE_INTERACTIVE | MODULE_RUN;
70 }
71
72
73 // ================================================
74 // == operator()                                    ==
75 // ================================================
76
77 int MyOwnModuleClass::operator() (TOCCLObject *Arg) {
78   TParameterList *Param;
79   TOCCLLong      Value;
80   char           String[256];
81
82   // Arg ist gleich Null, wenn das Modul gleich nach dem Laden
83   // aufgerufen wird.
84   if (Arg == NULL) {
85     // Ausgabe einer Meldung
86     ShowMessage
87       ("MyOwnModuleClass::operator() with Arg == NULL");
88     return MODULE_RETURNCODE_NOERROR;
89   }
90
91   // Ist der Parameter vom Typ TParameterList ?
92   if (dynamic_cast<TParameterList*>(Arg) != NULL) {
93     Param = dynamic_cast<TParameterList*>(Arg);
94
95     // Die Methoden von TParameterList werfen eine Exception,
96     //  wenn der angefoderte typ nicht passt. Daher eine
97     // try-catch Umgebung.
98     try {
99       Value = Param->GetLongParam (1);
100    }
101    catch (TOPERAEvent &Error) {
102      ShowMessage ("Wrong Parameter");
103      return MODULE_RETURNCODE_UNDEFERROR;
104    }
105    // Ausgabetext aufbereiten
106    sprintf (String, "Value: %li", (long)Value);
107
108    // Textausgabe über eine Dialogbox
109    ShowAlert (String);
110
111    return MODULE_RETURNCODE_NOERROR;
112  }
113
114  // Fehler, da kein bekannter Parameter
115  return MODULE_RETURNCODE_UNDEFERROR;
116 }
117
118
119 // ================================================
120 // == Edit                                          ==
```

```
121 // ================================================
122
123 TOCCLObject* MyOwnModuleClass::Edit (TOCCLObject *Arg) {
124   TParameterList *Param;            // Zeiger auf Parameterliste
125   Fl_Window      *Dialog;           // Fensterelemente
126   Fl_Button      *OkButton;
127   Fl_Button      *CancelButton;
128   Fl_Widget      *Button;
129   Fl_Input       *Text;
130   char           String[256];
131   long           Value;
132
133   // Wenn Arg gleich Null ist, dann muss Speicher für ein
134   // Parameterobjekt reserviert werden, sonst ist schon eins vorhanden.
135   if (Arg == NULL) {
136     Param = new TParameterList;
137     (*Param) << 0l;
138   }
139   else {
140     Param = dynamic_cast<TParameterList*>(Arg);
141   }
142
143   // Dialogbox erstellen und auf das Beenden des Dialogs warten
144   Dialog       = new Fl_Window (320, 200, "TPause");
145   OkButton     = new Fl_Button ( 20,160,80,30, "OK");
146   CancelButton = new Fl_Button (120,160,80,30, "Cancel");
147
148   Text = new Fl_Input (100,10, 200, 30, "Value:");
149
150   sprintf (String, "%li", (long)Param->GetLongParam (1));
151   Text->value(String);
152
153   Dialog->end();
154   Dialog->show();
155   do {
156     Fl::wait();
157     Button = Fl::readqueue();
158   } while ((Button != OkButton) && (Button != CancelButton));
159
160   // Wurde der Dialog abgebrochen ?
161   if (Button == CancelButton) {
162     if (Arg == NULL) {
163       delete Param;
164       Param = NULL;
165     }
166     delete Dialog;
167     return Param;
168   }
169
170   // Inhalt des Parameterobjektes ändern.
171   sscanf (Text->value(), "%li", &Value);
172   Param->SetParameter (1, Value);
173
174   // Dialogbox löschen
175   delete Dialog;
176
177   return Param;
178 }
179
180
181 // ================================================
182 // == GetTextFromParamString                     ==
183 // ================================================
184
185 void MyOwnModuleClass::GetTextFromParamString (TOCCLObject *Arg,
```

```
186                                                 char        *Text){
187   TParameterList *Param;
188   TOCCLLong      Value;
189
190   if (dynamic_cast<TParameterList*>(Arg) != NULL) {
191     Param = dynamic_cast<TParameterList*>(Arg);
192
193     try {
194       Value = Param->GetLongParam (1);
195     }
196     catch (TOPERAEvent &Error) {
197       ShowMessage ("Wrong Parameter");
198       return;
199     }
200     sprintf (Text, "(%li)", (long)Value);
201     return;
202   }
203 }
204
205
206 // ================================================
207 // == GetParameterDescription                    ==
208 // ================================================
209
210 TOCCLString MyOwnModuleClass::GetParameterDescription(void) {
211   return "Integerzahl";
212 }
```

Wie an den Methoden **`operator()`** und **`Edit`** zu sehen ist, existieren weiter Methoden und Klassen, die für die Programmierung eines Moduls hilfreich sind. Diese werden weiter unten erläutert.

Roboter-Module

Wie in Kapitel 3.3.1 beschrieben, werden die über *OPERA* kontrollierten Roboter nicht direkt angesteuert, sondern über eine Zwischenschicht, die die Befehle an die virtuellen auf die realen Roboter umsetzt. Diese Zwischenschicht wird über ein *Roboter-Modul* verwaltet, welches eine von **`TRunModule`** abgeleitete Klasse ist und die um die notwendigen Methoden erweitert wurde. Nur die im folgenden aufgeführten Methoden müssen überladen werden[6].

`GetRobotCount` liefert die Anzahl von dem Modul zur Verfügung gestellten Roboter.

```
int GetRobotCount (void) = 0;
```

`GetRobotName` liefert den Namen des Roboters mit dem Index **`RobotIndex`**. Ist der Index größer als von **`GetRobotCount`** zurückgeliefert, so ist der Rückgabewert Null.

```
virtual char* GetRobotName  (int RobotIndex)  = 0;
```

[6]Eine vollständige Beschreibung der Klasse **`TRobotModule`** findet sich auf der beiliegenden CD-ROM.

GetRobot liefert eine Instanz auf den Roboter mit dem Index **RobotIndex**.

```
TRemoteRobot* GetRobot (int RobotIndex) = 0;
```

Der Rückgabewert ist eine Instanz, die von der Klasse **TRemoteRobot** abgeleitet ist. Diese weiter unten beschriebene Klasse setzt die abstrakten Roboterkommandos in reale Anweisungen um.

StartRobots: Startet die Roboter.

```
virtual void StartRobots (void) = 0;
```

StopRobots: Stoppt die Roboter und gibt sie frei.

```
virtual void StopRobots (void) = 0;
```

RobotStarted: Die Methode gibt Auskunft, ob die Roboter gestartet worden sind. Sind die Roboter gestartet worden, so ist der Rückgabewert Null, sonst Eins.

```
virtual int RobotStarted (void) = 0;
```

A.2.1 Häufig benötigte Klassen

Um ein *OPERA*-Modul zu schreiben, welches nicht nur isoliert für sich eine Aufgabe erledigen, sondern welches auch mit dem Gesamtsystem in Verbindung treten kann, existieren einige Klassen, derer man sich bedienen sollte[7].

TVariableDatabase

Die Klasse ist dazu geeignet, beliebige abgeleitete Instanzen der Klasse **TOCCLObject** unter einem eindeutigen Namen zu speichern. Die wichtigsten Methoden sind:

Konstruktoren / Destruktoren Für diese Klasse stehen der Standardkonstruktor/ -destruktor und der *Copy*-Konstruktor zur Verfügung.

```
TVariableDatabase (void);
TVariableDatabase (const TVariableDatabase&);
~TVariableDatabase (void);
```

GetData Diese Methode liefert das Object mit dem Namen **Name** zurück. Sollte kein Objekt unter dem Namen gespeichert worden sein, ist das Ergebnis Null.

[7] Eine vollständige Auflistung aller zur Programmierung verwendeten Klassen und ihrer Methoden findet sich auf der beiliegenden CD-ROM.

```
TOCCLObject* GetData (const char *Name);
```

StoreData Speichert die Daten **Data** unter dem Namen **Name** ab. Ist eine Variable unter dem Namen schon vorhanden, so wird diese ersetzt.

```
void StoreData (const char *Name, TOCCLObject* Data);
```

DeleteData Löscht die Daten mit dem Namen **Name**.

```
void DeleteData (const char *Name);
```

GetFirstVariable Liefert die erste gespeicherte Variable.

```
TOCCLObject* GetFirstVariable (void);
```

GetNextVariable Liefert die Variable, die der Variable **Data** folgt. Mit den beiden Methoden **GetFirstVariable** und **GetNextVariable** können die gesamten gespeicherten Daten durchgegangen werden. Da die Möglichkeit besteht, dass während eines Durchlaufs ein zweiter Thread die Reihenfolge der Variablen ändert, sollte mit den Methoden **SetGlobalLock** und **ClearGlobalLock** die Instanz gesperrt werden.

```
TOCCLObject* GetNextVariable(TOCCLObject *Data);
```

GetName Liefert den Namen zu der Variablen **Data**. Ist eine Variable **Data** nicht gespeichert, so ist die Länge des zurückgelieferten String Null.

```
TOCCLString GetName (TOCCLObject *Data);
```

operator= Weist der Instanz **Right** zu.

```
void operator= (TVariableDatabase &Right);
```

operator== / **operator!=** Mit diesen beiden Operatoren kann die Gleich- oder Ungleichheit von zwei Instanzen abgefragt werden.

```
int operator== (TVariableDatabase &Right);
int operator!= (TVariableDatabase & Right);
```

Das in Kapitel 3.2 beschriebene *Blackboard* ist über eine solche Klasse realisiert. Da diese spezielle Instanz an vielen Stellen des System benötigt wird, wird sie als globale Variable **VarDatabase** exportiert.

```
extern TVariableDatabase VarDatabase;
```

TModuleDatabase

Eine Instanz dieser Klasse verwaltet alle Module, die zum System *hinzugeladen* werden. Folgende Methoden dieser Klasse werden nicht nur intern benötigt, sie sind auch für die Programmierung von Modulen wichtig. Über die globale Zeigervariable **ModuleDatabase** kann auf die Moduledatenbank von *OPERA* zugegriffen werden.

```
extern TModuleDatabase *ModuleDatabase;
```

GetInstance Diese Methode liefert die Instanz eines Moduls zu einem gegebenen Namen. Diese Information wird benötigt, wenn aus einem Modul ein anderes *direkt* aufgerufen wird. Der Parameter **Name** bezeichnet hierbei den Dateinamen des Moduls, nicht den Namen unter dem das Modul dem Gesamtsystem bekannt ist.

```
TRunModule* GetInstance (const char* Name);
```

GetInstanceFromCommandName liefert die Instanz eines Moduls in Abhängigkeit vom Kommandonamen **CommandName**.

```
TRunModule* GetInstanceFromCommandName (const char* CommandName);
```

Folgendes Beispiel zeigt die Verwendung von **GetInstanceFromCommandName**.

```
1   TRunModule     *Grabber;
2   TParameterList Param;
3   int            ReturnCode
4
5   Grabber = ModuleDatabase->GetInstanceFromCommandName ("Grabber");
6   if (!Grabber) {
7     ShowAlert ("No Grabber Module");
8     return -1;
9   }
10
11  ...
12
13  ReturnCode =  (*Grabber)(Param);
```

TParameterList

Diese Methode dient auch zur Aufnahme beliebiger Instanzen von Klassen, die von TOCCLObject abgeleitet worden sind. Sie werden sortiert abgelegt, so dass über einen konstanten Index immer auf dieselbe Variable zugegriffen werden kann. Neben dem Stand-, *Copy*-Konstruktor und Destruktor sind folgende Methoden von Interesse:

AddParameter Über diese Methoden wird eine Variable in der Liste gespeichert.

```
void AddParameter (TOCCLString Name,TOCCLObject* Ptr,int Flag=0);
void AddParameter (const char* Name);
void AddParameter (long Value);
void AddParameter (float Value);
```

Name ist ein frei zu wählender Name für die zu speichernde Variable. **Ptr** ist ein Zeiger auf die zu speichernde Funktion, wobei **Flag** angibt, ob die Variable beim Löschen der Parameterstruktur auch gelöscht werden soll oder nicht. Die Methode mit nur einem Parameter dient einfach dazu, Standarddatentypen in einer Parameterstruktur zu speichern, ohne vorher selbst die entsprechenden *OCCL*-Datenstrukturen anlegen zu müssen. Die Daten werden immer ans Ende der Liste gehängt.

GetParamCount liefert die Anzahl der gespeicherten Daten.

```
int GetParamCount(void);
```

GetParameter liefert die zum Index **Index** entsprechende Variable. Der zweite Parameter gibt den Typ der gewünschten Variablen an. Stimmt der gewünschte Typ nicht mit dem Type der Variable an der Position **Index** überein, so wird eine Ausnahme ausgelöst.

SetParameter setzt den Parameter an der Position **Index**.

```
void SetParameter (int Index, long  Long);
void SetParameter (int Index, float Float);
void SetParameter (int Index, char*);
```

TRemoteRobot

Diese Klasse dient dazu, einen Satz von abstrakten Roboterbefehlen zur Verfügung zu stellen und diese in Kommandos für die jeweiligen realen Roboter umzusetzen, um die Ansteuerung für einen realen Roboter zu implementieren. Hierfür muss eine eigene Klasse von der Klasse **TRemoteRobot** abgeleitet werden und alle folgenden Methoden überdefiniert werden.

Konstruktor / Destruktor: Für die Klasse existiert nur der Standardkon- und destruktor.

```
TRemoteRobot (void);
~TRemoteRobot (void);
```

Stop Robot / GetMotionAbortFlag / ClearMotionFlag: **StopRobot** bewirkt den Abbruch der aktuellen und aller gepufferten Bewegungen. Dies wird üblicherweise dadurch ausgelöst, dass ein Flag gesetzt wird, welches der Kontrollschicht signalisiert,

dass die Bewegung abgebrochen werden soll. Der Status dieses Flags kann mit der Methode **`GetMotonAbortFlag`** abgefragt und mit **`ClearMotionAbortFlag`** wieder auf Null zurückgesetzt werden.

```
void StopRobot(void);
int  GetMotionAbortFlag   (void);
void ClearMotionAbortFlag (void);
```

GetRobotName liefert den Namen des Roboters zurück. Der Name wird im Parameter **`Name`** gespeichert.

```
void GetRobotName (char* Name);
```

SetBase / SetTool setzt die Basis- bzw. Tooltransformation des Roboters.

```
void SetBase (const TTransform &Base);
void SetTool (const TTransform &Tool);
```

GetBase / GetTool liefert die gesetzte Basis- bzw. Tooltransformation des Roboters.

```
TTransform GetBase (void);
TTransform GetTool (void);
```

SetStdSpeed / GetStdSpeed `GetSpeedSpeed` setzt die Geschwindigkeit des Roboters, mit der er standardmäßig Bewegungen abfährt. Mit **`GetStdSpeed`** kann die eingestellte Geschwindigkeit abgefragt werden. Dieser Wert wird immer dann verwendet, wenn die zu fahrende Geschwindigkeit nicht explizit angegeben wurde.

```
void  SetStdSpeed (float);
float GetStdSpeed (void);
```

SetStdMod / GetStdMod: `SetStdMode` setzt den Interpolationsmodus, mit dem der Roboter den Endpunkt anfahren soll. Mit **`GetStdMod`** kann man diesen Wert abfragen. Dabei bedeutet ein *c* karthesische Interpolation, d.h. eine gradlinige Bewegung im kartesischen Raum, und *j* eine gradlinige Bewegung im Konfigurationsraum.

```
void  SetStdMod   (char);
char  GetStdMod   (void);
```

SetMotionQueueSize setzt die Größe des Bewegungspuffers. In Abhängigkeit von der Größe dieses Puffers können Bewegungen initiiert werden, ohne auf das Ende einer vorherigen Bewegung warten zu müssen.

```
int SetMotionQueueSize (int);
```

OpenHand / CloseHand / GripperState: **`OpenHand`** und **`CloseHand`** öffnet bzw. schließt die Hand des Roboters. Mit **`GripperState`** kann der momentane Zustand der Hand abgefragt werden. Der zurückgelieferte Wert ist Eins, wenn die Hand offen ist, sonst Null.

```
void OpenHand (void);
void CloseHand (void);
int  GripperState(void);
```

StopCurrentMotion stoppt die aktuelle Bewegung. Sind noch weitere Bewegungen gepuffert, so werden diese sofort ausgeführt.

```
void StopCurrentMotion (void);
```

WaitForCompleted wartet auf die Beendigung der aktuellen und aller gepufferten Bewegungen.

```
void WaitForCompleted (void);
```

StdMove führt eine einfache Bewegung aus. Interpolationsmodus und Geschwindigkeit sind abhängig von den mit **`SetStdMod`** und **`SetStdSpeed`** gesetzten Werten. Der Parameter **`Name`** kann frei gewählt werden.

```
TError StdMove (char *Name, const TTransform &Transform);
TError StdMove (char *name, const TTransform *Transform);
```

Move Im Gegensatz zu StdMove kann bei der Methode **`Move`** der Interpolationsmodus der Bewegung über den Parameter **`Mod`** explizit angegeben werden.

```
TError Move (char *Name, const char Mod, const TTransform &Transform);
```

Movej bewegt den Roboter zu der über **`Joints`** angegebenen Gelenkwinkelstellung. Die Bewegung erfolgt geradlinig im Konfigurationsraum.

```
TError Movej (const TJoints &Joints);
```

GetMaxJoint / GetMinJoint liefert den maximalen bzw. minimalen Gelenkwinkel für das über **`JointNo`** bezeichnet Gelenk.

```
TAngle GetMaxJoint (int JointNo);
TAngle GetMinJoint   (int JointNo);
```

GetJoint6 liefert die aktuellen 6 Gelenkwinkel des Roboters.

```
TJoints GetJoint6 (void);
```

Gett6Transfrom / GetTCP **`Gett6Transform`** liefert die aktuelle *T6* Transformation des Roboters, während **`GetTCP`** den aktuellen *Tool Center Point* liefert.

```
TTransform Gett6Transform  (void);
TTransform GetTCP          (void);
```

s_rotateZRel rotiert das Tool um den Annäherungsvektor, um den in **`Angle`** angegeben Winkel mit der Geschwindigkeit **`Speed`**.

```
TError s_rotateZRel (const TAngle &Angle, const float Speed);
```

GetForce*x* / **GetTorque*x*** liefert die entsprechenden Kräfte und Drehmomente, die am Tool anliegen.

```
float GetForceN  (void);
float GetForceO  (void);
float GetForceA  (void);
float GetTorqueN (void);
float GetTorqueO (void);
float GetTorqueA (void);
```

s_move*x*Rel bewegt das Tool entlang des entsprechenden Toolvektors mit der angegebenen Geschwindigkeit und dem Interpolationsmodus.

```
TError s_moveNRel (const float Dist,       const float Speed,
                   const float AbortForce, char  Mode = 'c');
TError s_moveORel (const float Dist,       const float Speed,
                   const float AbortForce, char Mode = 'c');
TError s_moveARel (const float Dist,       const float Speed,
                   const float Speed,      char Mode = 'c');
```

adjust regelt die Kräfte und Drehmomente auf die angegebenen Werte ein. **`Mask`** ist eine Bitmaske, die angibt, welche Kräfte und Drehmomente eingeregelt werden sollen.

```
TError adjust (const float ForceN,  const float ForceO,  const float ForceA,
               const float TorqueN, const float TorqueO, const float TorqueA,
               int Mask);
```

Für die entsprechenden Bits existieren die Makros **`FORCE_N, FORCE_O, FORCE_A, TORQUE_N,TORQUE_O,TORQUE_A`**, die bei Bedarf bitweise verodert werden.

SpecialCommand dient zur Übertragung von roboterspezifischen Befehlen (z.B. das Kalibrieren von Sensoren). Ein solcher Befehl besteht aus einer Befehlsnummer, einer Unterbefehlsnummer und untypisierten Daten, deren Länge anzugeben ist.

```
TError SpecialCommand(int  CommandNo,    int SubCommandNo,
                      void *ParamBuffer, int BufferSize)
```

A.2.2 Threads

OPERA verwendet mehrere Threads, um den internen Ablauf zu koordinieren. Die unterschiedlichen Methoden eines Moduls werden daher von unterschiedlichen Threads aufgerufen, worin der Grund liegt, dass Funktionen, die auf die Funktionen von *X-Windows* basieren, nur in bestimmten Methoden aufgerufen werden dürfen. Ein Modul kann aber auch ein oder mehrere Threads starten. Ein Beispiel hierfür ist ein Sensormodul, welches in festen Abständen seine Daten im Sensorfenster aktualisieren möchte.

Damit verschiedene Threads auf den gleichen Datenstrukturen arbeiten können, muss die Möglichkeit gegeben sein, diese zu sperren. Die verschiedenen Listen- bzw. Datenbankklassen sind von sich aus *Multithread-Safe*. Die Datenstrukturen, die sie verwalten, sind es nicht. Wird daher auf eine Datenstruktur zugegriffen, die auch von anderen Threads verwendet wird, ist diese über ein Mutex zu schützen. Mit der Klasse **`TOCCLMutex`** steht eine entsprechende Klasse zur Verfügung. Klassen, die von **`TOCCLObject`** abgeleitet worden sind, besitzen drei Methoden, über die ein Sperren erfolgen kann: **`TOCCLObject::Lock`**, **`TOCCLObject::Trylock`** und **`TOCCLObject::Unlock`**. Es muss daher für Instanzen dieser Klassen kein weiterer Mutex angelegt werden.

A.2.3 Übersetzen

Um *OPERA* zu übersetzen muss die Datei `opera.tar.gz` in einem beliebigen Verzeichnis ausgepackt werden.

```
user@rechner> tar -xzf opera.tgz
```

Die notwendigen Dateien werden in verschiedene Unterverzeichnisse kopiert, die folgenden Inhalt haben:

DOC: Klassendokumentation in HTML;

MATHE: Einfache Matrix- und Vektorklasse;

MODULE: Klassen der *OPERA*-Module;

OCCL: Klassen der *OCCL*-Bibliothek;

OPERA: Klassen für *OPERA*;

RCCL-OCCL: Wrapperklassen für RCCL;

RCCLSIM: Simulationsklassen, als Ersatz für RCCL;

UTILS: Allgemeine Hilfsklassen;

include: Header-Dateien;

lib: Übersetzte Bibliotheken.

Über das Kommando

```
user@rechner> make clean depend all
```

werden alle Klassen, Programme und Module übersetzt. Alle übersetzten Bibliotheken befinden sich anschließend im Verzeichnis `lib` und die ausführbare Binärdatei von *OPERA* unter `bin`. Um das Programm ausführen zu können, muss der `lib`-Pfad im Suchpfad für Bibliotheken vorhanden sein. Hierzu muss der Pfad in die Umgebungsvariable `LD_LIBRARY_PATH` aufgenommen werden.

Die GNU Compiler Collection in der Version 2.95.2 ist zum Übersetzen verwendet worden. Dies ist der C/C++ Compiler aller aktuellen Linux-Distributionen. Frühere GCC Versionen können nicht verwendet werden, da in diesen das *Exception-Handling* nicht *Multithread-Safe* ist.

Benötigte Fremdsoftware

Um *OPERA* komplett übersetzen zu können, müssen folgende Programmpakete vorhanden sein:

FLTK *OPERA* nutzt für den Aufbau seiner Oberfläche das Programmpaket *FLTK*. Es findet sich unter `http://fltk.org`. Das Paket sollte unter `/vol/fltk` installiert sein, um Änderungen in den *Makefiles* zu vermeiden. Die Bibliotheken sollten unter `/vol/fltk/lib` und die Header-Datei unter `/vol/fltk/include` stehen.

Magic Für die Übersetzung des Moduls **`Grabber.dll`** muss die Magic-Bibliothek von Christian Scheering[Sch00a] installiert sein. Um Änderungen in den *Makefiles* zu vermeiden, sollten die Bibliotheken unter `/vol/magic/lib` und die Header-Dateien unter `/vol/magic/include` zu finden sein und die folgenden symbolischen Links im Verzeichnis **`/vol/ti/include`** oder im Include-Pfad von *OPERA* existieren:

```
grab++    -> /vol/magic/demos/imgserver/src/grab++/
imgserver -> /vol/magic/demos/imgserver/src/include/
pca       -> /vol/magic/demos/eigen1/src/
tools     -> /vol/magic/src/tools/
```

Andere Zusätzlich werden folgende Bibliotheken benötigt, die jeder Standard-Linuxinstallation beiliegen: `libz.so, libGL.so`.

A.2.4 Portierung auf andere Robotersteuerungen

Die ursprüngliche Version von *OPERA* setzt auf die Robotersteuerungsbibliothek *RCCL*[HP86] und der darauf aufsetzenden Klassenbibliothek *RCCL++* auf. Die beiliegende Version von *OPERA* ist daher dahingehend modifiziert worden, die von *RCCL* und *RCCL++* abhängigen Teile durch Platzhalterklassen zu ersetzen. Soll *OPERA* auf andere Software portiert werden, so sind die folgenden Klassen an die gewünschte Robotersteuerung anzupassen[8].

`tjoints`: Diese Klasse repräsentiert Gelenkwinkelkonfigurationen eines Roboters. Bei einem 6-gelenkigen Roboter besteht sie aus 6 Fließkommazahlen für die Gelenkwinkel.

`ttransform` bildet eine homogene Transformation ab und besteht üblicherweise aus einer 4×4 Matrix.

`tvector`: Die Klasse stellt einen dreidimensionalen Vektor zur Verfügung.

`TRemoteRobot`: Diese Klasse dient, wie zuvor beschrieben, als Schnittstelle zwischen *OPERA* unter der realen Robotersteuerung. Es muss daher eine entsprechende Roboterklasse erstellt werden, die von **`TRemoteRobot`** abgeleitet worden ist und deren Methoden überlädt.

[8] Eine genaue Beschreibung des Funktionsumfanges befindet sich auf der beiliegenden CD-ROM.

Anhang B

OCCL

B.1 Ziel der Library

Um die Flexibilität von *OPERA* zu erreichen, wird an vielen Stellen des Systems mit Instanzen von Datentypen gearbeitet, ohne dass deren genauer Typ bekannt ist. Zusätzlich müssen diese Datentypen leicht in Datenströme umgewandelt werden können, da sie des öfteren über Datenkanäle *verschickt* werden müssen. Daher ist zum einen notwendig, einen Grunddatentyp (bzw. Klasse) als Basis für die Ableitunghierarchie und zum anderen ein Konzept, diese Datentypen in einen seriellen Datenstrom umzuwandeln zu besitzen.

Zu diesem Zweck wurde *OCCL* (Open Communication C++ Library) entwickelt. Die Library hat das Ziel, den Programmierer bei der Übertragung von Daten zwischen zwei Prozessen zu unterstützen und gleichzeitig Basisdatentypen für eigene Datenstrukturen bzw. Klassen zur Verfügung zu stellen . Es existieren Mechanismen, die es einem Anwender der Library erlauben, eigene Datenstrukturen ohne großen Programmieraufwand zwischen mehreren Prozessen auszutauschen. Dabei ist es unerheblich, ob diese Prozesse auf demselben oder auf unterschiedlichen Rechnern (-architekturen) ablaufen. Die Library ist deshalb architekturunabhängig und kümmert sich um die richtige Konvertierung zwischen den einzelnen *Endian*-Formaten unterschiedlicher Rechnerplattformen. Damit die Library einfach zu verwenden ist und keine speziellen Anforderungen an den Progammaufbau stellt, wurden die notwendigen Methoden in die Basisklasse integriert und dem Programmierer zum Programmieren von abgeleiteten Klassen nur wenig Zusatzarbeit aufgebürdet. Es werden einfache Mechanismen der Sprache C++ verwendet, die auch von Dritten nachvollzogen werden können. Der Einsatzbereich ist nicht spezialisiert, sondern die Library kann auf unterschiedlichen Gebieten zum Einsatz kommen. Es besteht keine Festschreibung auf feste Datentypen, sondern es können jederzeit eigene Klassen und Strukturen verwendet werden.

Um diesen Anforderungskatalog ohne zusätzliche Präprozessoren / Metacompiler zu realsieren, wurde als Sprache C++ (ISO/IEC 14882) verwendet, wobei sowohl *Exception-Handling* und *Runtime-Type-Information* verwendet wurden.

B.2 Überblick über die Realisierung

Alle Klassen und Strukturen werden über den Stream-Mechanismus[1] von C++ versendet oder empfangen. Damit die Streams von *OCCL* nicht mit den Standardstreams von C++ verwechselt werden, werden sie hier als Kanäle bezeichnet. Ist z.B. ein Objekt **`TMyObject`** definiert, welches versendet werden soll, so lautet der Quelltext z.B.:

```
1  // Definition einer Instanz der Klasse TMyObject
2  TMyObject A;
3
4  // Definition eines Kanals der Klasse TMyChannel
5  TChannel  MyChannel;
6
7  // Versenden der Instanz
8  MyChannel << A;
9
10 // Empfangen einer Instanz
11 MyChannel >> A;
```

Es können ebenfalls Datenstrukturen empfangen werden, deren genauer Typ nicht vorher bekannt ist. Die Datenstruktur muss nur von der von *OCCL* definierten Basisklasse `TOCCLObject` abgeleitet sein und sowohl dem Sender als auch dem Empfänger bekannt sein. Die Datenstrukturen sähe z.B. so aus:

```
1  TOCCLObject *ObjectPointer;  // Zeiger auf den Typ der Basisklasse
2  TChannel    MyChannel;
3
4  MyChannel >> &ObjectPointer;
```

OCCL kreiert eine entsprechende Instanz der ankommenden Datenstruktur und speichert die Adresse der empfangenen Datenstruktur in der angegebenen Zeigervariable (hier **`ObjectPointer`**).

Über die Kanäle wird der Vorgang (die Programmierung) des Versendens und des Empfangens gegenüber dem Übertragungsmedium (Pipe, shared Memory, Sockets, Files) transparent. Die Kanäle können auch für sich alleine eingesetzt werden. Man kann also nicht nur Strukturen oder Klassen, sondern auch unstrukturierte Daten über sie versenden (z.B. einen zusammenhängenden Speicherbereich oder einen Integer).

Die Grundvoraussetzung für die Art der Übertragung ist, die Fähigkeit einer Klasse sich zu serialisieren. Um diesen Mechanismus auf alle Objekte anwenden zu können, werden die Objekte, die zwischen Prozessen übertragen werden sollen, von der Basisklasse abgeleitet. Diese Basisklasse enthält alle grundlegenden Mechanismen, um eine von ihr abgeleitete Klasse zu serialisieren. Eine neu hinzukommende Klasse, welche wiederum nur Objekte beinhaltet, die von der Basisklasse abgeleitet sind, muss nur bei einer zentralen Verwaltungsinstanz registriert werden. Es müssen keine weiteren Methoden zum Versenden und Empfangen programmiert werden. Erst wenn eine neue Klasse komplexe Datenstrukturen beinhaltet, müssen eigene Methoden zum Versenden und Empfangen der Klasse selbst geschrieben werden.

[1] $>>$ und $<<$ Operatoren

B.3 Beispielquelltext

```
#include <iostream.h>
#include <OCCL/TOCCLObjects.H>

class MyOwnClass : public TOCCLObject {
protected:
  TOCCLLong  Long;                        // OCCL- Datentyp für Long
  TOCCLFloat Float;                       // OCCL- Datentyp für Float
  char       Text[256];                   // Zeichenkette
public:
  MyOwnClass (void) {Long = 0; Float = 0.0f;}  // Konstruktor
  virtual ~MyOwnClass (void);                   // Destruktor

  virtual int Read  (TOCCLChannel &Channel);  // Methode zum Empfangen
  virtual int Write (TOCCLChannel &Channel);  // Methode zum Versenden
  virtual int GetSize (void);                 // Liefert die Groesse
                                              // der zu versendenden
                                              // Daten
  RTTI(MyOwnClass)                       // OCCL- Macro zur
                                         // definition zusaezl.
                                         // Methoden
};


MyOwnClass::~MyOwnClass (void) {
}

int MyOwnClass::Read (TOCCLChannel& ReadChannel) {
  int Changed;

  Changed = TOCCLObject::Read (Channel); // Empfangen der Daten von
                                         // Vorgaengerklasse. Ist der
                                         // Inhalt von Changed != NULL
                                         // so haben die empfangenen
                                         // Daten den falschen Endian.

  Long.Read (Channel);                   // Empfangen der Daten vom "Long"
  Float.Read (Channel);                  // Empfangen der Daten vom "Float"

  Channel.Read (256, Text);

  return Changed;
}

int MyOwnClass::Write (TOCCLChannel& ReadChannel) {
  TOCCLObject::Write (Channel);   // Versenden der Daten von Vorgaengerklasse
  Long.Write (Channel);           // Versenden der Daten vom "Long"
  Float.Write (Channel);          // Versenden der Daten vom "Float"

  Channel.Write (256, Text);

  return 0;
}

int MyOwnClass::GetSize (void) {
  return TOCCLObject::GetSize() + Long.GetSize() + Float.GetSize() + 256;
}


int main (void) {
  TFileChannel Channel("/tmp/Test.dat", "w+");  // Kanal fuer eine Datei
  MyOwnClass   MyInstance1;                     // Zwei Instanzen der
```

```
62   MyOwnClass   MyInstance2;                    // eigenen Klasse
63
64   MyInstance1.Register();                     // Klasse registrieren !!!
65
66   Channel << MyInstance1;                     // Versenden
67   Channel >> MyInstance2;                     // Empfangen
68
69   return 0;
70 }
```

Die Zeilen 4-21 zeigen eine einfache Beispielklasse, die aus einem **TOCCLLong**, einem **TOCCLFloat** und einer Zeichenkette besteht. Neben den Methoden **Read**, **Write** und **GetSize** (Zeile 13 -15), darf das Makro **RTTI** (Zeile 18) nicht vergessen werden, welches einige benötigte Methoden in die Klasse einfügt. In der Implementierung der **Read-** und **Write-** Methode müssen immer zuerst die **Read-** und **Write-**Methoden der Vorgängerklasse aufgerufen werden (Zeile 30 bzw. 45). Die Methode **GetSize** (Zeile 54-56) errechnet die Größe der zu versendenen Daten. Die Größe ist die Größe der Vorgängerklasse plus der Größe der Variablen **Long**,**Float** und **Text**. Bevor eine Instanz einer Klasse versendet werden kann, muss sie registriert sein (Zeile 64). Ist sie es nicht, so wird eine Exception vom Typ **TOCCLException** geworfen.

B.4 Einschränkungen

- Es können nicht vollständig unbekannte Klassen versendet bzw. empfangen werden. Eine Klasse, die versendet werden soll, muss auf beiden Seiten des Kommunikationskanals bekannt, d.h. registriert sein. D.h. es besteht kein Mechanismus zur Übertragung der Semantik eines Objekts, wie z.B. in Java.
- Es gibt bestimmte Einschränkungen in der Vererbbarkeit in der Klassenhierarchie.

Anhang C

Anwendung der Reduktionsverfahren

PCA-Berechnung

Für die Reduktion von Sensordaten über die PCA wurde u.a. die Klasse **`eigen`** verwendet. Die Klasse ist einfach aufgebaut und enthält neben dem Konstruktor und dem Destruktor nur eine öffentliche Methode **`berechnung`** und zwei öffentliche Variablen **`mat`** und **`eigenwerte`**.

```
1  class eigen {
2  private:
3    vektor<float> subdiagonale;
4
5    // Householder-Reduktion bis zur Tridiagonalform
6    void Tridiagonal2 ();  // schnelle Spezialversionen
7    void Tridiagonal3 ();  // für kleine Matrixgrößen
8    void Tridiagonal4 ();
9    void TridiagonalN ();
10
11   // QL Algorithmus für tridiagonalisierte Matritzen
12   void QL();
13
14   unsigned int max_iterationen, zuviele_iterationen;
15
16 public:
17   eigen (unsigned int mi = 30){
18     max_iterationen    = mi;
19     zuviele_iterationen= 0;
20   }
21   ~eigen (void){}
22
23   quad_matrix<float> mat;
24   vektor<float>      eigenwerte;
25
26   void berechnung(void);
27 };
```

Alle anderen Methoden dieser Klasse sind privat und dienen zur Berechnung der *pricipal components*. Anhand des folgenden Beispiel soll die Verwendung dieser Klas-

se erläutert werden[1]. Es sollen die Eigenwerte und Eigenvektoren einer 2×40 Matrix berechnet werden, wobei die Matrixelemente als 40 zweidimensionale Mustervektoren angesehen werden. Die Muster werden als Spaltenvektoren in der Matrix abgelegt.

```
1  #define DIM 2
2  #define ANZ_V 40
3
4  int main (void)
5  {
6    eigen         eig;
7    matrix<float> X(DIM, ANZ_V);
8    vektor<float> durch(DIM);
9    unsigned int  i;
10   unsigend int  j;
11
12   durch[0] =
13   durch[1] = 0.0f;
```

Zu Beginn werden die benötigten Variablen angelegt und teilweise initialisiert. Dies ist eine Instanz der Klasse **`eigen`**, die Matrix für den Mustervektoren **`X`** und ein paar Hilfsvariablen.

```
1
2    X.zs(0,0) =5.7;   X.zs(1,0)=2.1;
3    X.zs(0,1) =5.1;   X.zs(1,1)=3.3;
4    X.zs(0,2) =5.2;   X.zs(1,2)=3.2;
5    X.zs(0,3) =5.3;   X.zs(1,3)=2.4;
6    X.zs(0,4) =5.4;   X.zs(1,4)=2.7;
7    X.zs(0,5) =5.5;   X.zs(1,5)=2.3;
8    X.zs(0,6) =5.6;   X.zs(1,6)=3.6;
9    X.zs(0,7) =5.7;   X.zs(1,7)=3.5;
10   X.zs(0,8) =5.8;   X.zs(1,8)=3.8;
11   X.zs(0,9) =5.9;   X.zs(1,9)=3.7;
12   X.zs(0,10)=5.2;   X.zs(1,10)=2.7;
13   X.zs(0,11)=6.1;   X.zs(1,11)=2.3;
14   X.zs(0,12)=6.2;   X.zs(1,12)=2.4;
15   X.zs(0,13)=6.3;   X.zs(1,13)=3.2;
16   X.zs(0,14)=6.4;   X.zs(1,14)=2.3;
17   X.zs(0,15)=6.5;   X.zs(1,15)=2.7;
18   X.zs(0,16)=6.6;   X.zs(1,16)=3.5;
19   X.zs(0,17)=6.7;   X.zs(1,17)=2.6;
20   X.zs(0,18)=6.8;   X.zs(1,18)=3.7;
21   X.zs(0,19)=6.9;   X.zs(1,19)=2.8;
22
23   X.zs(1,20)=6.8;   X.zs(0,20)=1.1;
24   X.zs(1,21)=7.8;   X.zs(0,21)=1.3;
25   X.zs(1,22)=7.3;   X.zs(0,22)=2.2;
26   X.zs(1,23)=6.9;   X.zs(0,23)=2.4;
27   X.zs(1,24)=6.1;   X.zs(0,24)=2.7;
28   X.zs(1,25)=7.5;   X.zs(0,25)=2.3;
29   X.zs(1,26)=7.2;   X.zs(0,26)=2.6;
30   X.zs(1,27)=6.7;   X.zs(0,27)=1.5;
31   X.zs(1,28)=6.3;   X.zs(0,28)=1.8;
32   X.zs(1,29)=7.6;   X.zs(0,29)=1.7;
33   X.zs(1,30)=6.7;   X.zs(0,30)=2.1;
34   X.zs(1,31)=6.2;   X.zs(0,31)=2.3;
```

[1]Dieses Beispiel und die Berechnung der PCA finden sich auf der beigelegten CD unter dem Verzeichnis `EIGEN/`.

```
35  X.zs(1,32)=7.4;   X.zs(0,32)=2.4;
36  X.zs(1,33)=7.5;   X.zs(0,33)=1.2;
37  X.zs(1,34)=6.9;   X.zs(0,34)=2.3;
38  X.zs(1,35)=7.9;   X.zs(0,35)=2.7;
39  X.zs(1,36)=6.2;   X.zs(0,36)=1.5;
40  X.zs(1,37)=7.2;   X.zs(0,37)=2.6;
41  X.zs(1,38)=7.7;   X.zs(0,38)=2.7;
42  X.zs(1,39)=7.1;   X.zs(0,39)=1.8;
```

Auf die einzelnen Elemente der hier verwendeten Matrixklasse wird über die Methode **`zs`** zugegriffen. Als Parameter wird erst die Zeile und dann die Spalte des Elementsangegeben. Nach der Initialisierung der Matrix wird der Durchschnittvektor berechnet und von allen Mustern abgezogen.

```
1  for (j=0;j<DIM;++j) {
2    for (i=0;i<ANZ_V;++i)
3      durch[j]+=X.zs(j,i);
4    durch[j]/=float(ANZ_V);
5  }
6
7  for (j=0;j<DIM;++j)
8    for (i=0;i<ANZ_V;++i)
9      X.zs(j,i)-=durch[j];
```

Es folgt die Berechnung der Kovarianzmatrix, die der Klassenvariablen **`eigen::mat`** abgelegt wird, und über den Aufruf der Methode **`eig::berechnung()`** die Berechnung der Eigenwerte und Eigenvektoren.

```
1  X.kovarianz(eig.mat); // Berechnen der Kovarianzmatrix
2
3  eig.berechnung();     // Berechnung der Eigenwerte und Vektoren
```

Die Eigenwerte sind danach in der Klassenvariablen **`eigen:eigenwerte`** und die Eigenvektoren in Variablen **`eigen::mat`** abgelegt. Die Eigenwerte und Vektor sind nicht der Größe nach sortiert.

Die Dimensionsreduzierung erfolgt durch den Aufbau einer entsprechenden Projektionsmatrix bei der die Zeilenvektoren aus den berechneten Eigenvektoren besteht.

```
1  matrix<float> Proj(1, DIM);
2
3  Proj.zs(0,0) = eig.mat(0,0);
4  Proj.zs(0,1) = eig.mat(1,0);
```

Die Projektion ergibt sich anschließend aus einer Matrix-Vektormultiplikation.

```
1  float Projektion;
2
3  Projektion = Proj.zs(0,0) * X.zs(16,0) + Proj.zs (0,1) * X.zs (16,1);
```

ORF-Berechnung

Im Gegensatz zur PCA wird bei der Berechnung der ORF die spätere Sollausgabe des zu bauenden Reglers berücksichtigt. Dazu wird der Fehler über die partielle Ableitung der Fehlerfunktion $(ys - y) * x[j])$ zurückpropagiert. Folgende Funktion berechnet die ORF[2].

```
1  #define ITERATIONEN 100000000
2
3  float CalcFeature (Matrix &TrainData, Vector &Soll, Vector &w) {
4    Vector     Proj     (TrainData.GetZeilen());
5    float      y;
6    float      Corr;
7    float      OldCorr  = 0.0;
8
9    w.Init();
10
11
12   for (int I = 0; I < ITERATIONEN; I++) {
13     for (int MusterIndex=0; MusterIndex<TrainData.GetZeilen(); MusterIndex++){
14       y = w.Scalar (TrainData[MusterIndex]);
15       w += (Soll[MusterIndex] - y) * TrainData.Zeile(MusterIndex);
16     }
17
18     if ((I & 255) == 0) {
19       Proj = TrainData * w;
20       Corr = fabs(CalcCorrelation (Proj, Soll));
21
22       if ((I & 1023) == 0) {
23         cout << "Index: " << I << "  Corr: " << Corr << endl;
24       }
25
26       if (Corr <= OldCorr)
27         break;
28
29       OldCorr = Corr;
30     }
31   }
32   w    = w.Norm();
33   Proj = TrainData * w;
34   Corr = fabs(CalcCorrelation (Proj, Soll));
35   return Corr;
36 }
```

Als Parameter (Zeile 3) erhält die Funktion die Trainingsmuster **`TrainData`**, die späteren Sollausgaben **`Soll`** und den zu berechnenden Projektionsvektor **`w`**. Zu Beginn (Zeile 4-8) werden die notwendigen lokalen Variablen angelegt. Der Dimension Vektor **`Proj`** ist dann gleich der Dimension eines einzelnen Mustervektors. Die einzelnen Muster sind Zeilenweise in der Matrix **`TrainData`** abgelegt. Nach dem Nullsetzen des Vektors **`w`** wird in den folgenden Schleifen (Zeile 14-36) die ORF berechnet, indem mehrmals die Projektion alle Mustervektoren über den Projektionsvektor **`w`** berechnet und die oben beschrieben Korrektur des Vektors durchgeführt wird. In den Zeilen (21-35) wird alle 256 Iterationen geprüft, ob sich die Korrelation zwischen der Projektion und den Solldaten ändert. Ändert sich die Korrelation nicht, so wird die Berechnung abgebrochen

[2]Der Quelltext befindet sich auf der beiliegenden CD-ROM unter **`opera/ORF`**.

und der Projektionsvektor aus 1 normiert (Zeile 36). Für die Berechnung der ORF sollten die Solldaten in das Intervall $[1, -1]$ projiziert werden. Die Trainingdaten müssen mittelwertsfrei sein. Das folgende Programm zeigt die Berechnung der ORF für die in Kapitel 4.5 verwendeten Daten.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <iostream.h>
4
5  #include "ORF/ORF.H"
6  #include "UTILS/TPGMFile.H"
7  #include "UTILS/global.H"
8  #include "MATHE/vector.H"
9  #include "OCCL/TOCCLObjects.H"
10
11 int main (int argc, char *argv[]) {
12   Matrix TestData1;  // Matrix für die Testmuster
13   Vector  TestSoll;  // Vektor für die Testsollausgaben
14   Matrix TrainData1; // Matrix für die Trainingsmuster
15   Vector  TrainSoll; // Matirx für die Trainingsolldaten
16
17   OCCLRegister();    // Registrieren der OCCL-Objecte
18   {
19     TPGMFile File;
20
21     File.Register();
22   }
23
24   // Lesen der Trainings- und Testdaten
25   ReadData (TrainData1, TrainSoll, TestData1, TestSoll);
26
27   // Projektion der Solldaten auf das Intevall[-1, 1]
28   SetVecInterval (Soll, -1.0, 1.0);
29
30   // Energienormaierung der Trainingsdaten
31   NormEnergie (TrainData1);
32
33   // Traingsdaten mittelwertfrei machen
34   SubAverage  (TrainData1);
35
36
37   Vector Proj (TrainSoll.GetSpalten());
38
39   // Berechnung des Projektionsvektors
40   CalcFeature (TrainData1, TrainSoll, Proj);
41
42   // Anzeigen des Projektionsvektors
43   Proj.Show();
44
45   return 0;
46 }
```

Nach dem Registrieren der Objekte, die von **`TOCCLObject`** abgeleitet worden sind, und dem Einlesen der Trainings- und Testdaten über die Funktion **`ReadData`** werden die Trainingssolldaten in das Intervall $[1, -1]$ projiziert (Zeile 28). Die Komponenten des Vektors haben damit einen minimalen Wert von -1 und einen maximalen Wert von 1. Da die Trainingsdaten Bilder sind werden sie energienormiert (Zeile 31). Hiernach werden die Daten mittelwertsfrei gemacht, indem der Durchschnittsvektor von allen Trainingsvektoren abgezogen wird (Zeile 34) und der Projektionsvektor berechnet (Zeile 43).

Anhang D

Petri-Netz

Nach [Eng88] ist ein Petri- Netz ein Modell zur Beschreibung und Analyse von Abläufen mit nebenläufigen Prozessen und nicht deterministischen Vorgängen. Sie eignen sich zur Beschreibung dynamischer Systeme, die eine feste Grundstruktur besitzen. Beispiele sind hierfür sind Rechenanlangen, Betriebssysteme, Organisationsabläufe und auch Montageanleitungen.

Ein Petri- Netz ist ein gerichteter Graph, der aus zwei verschiedenen Sorten von Knoten besteht: Stellen und Transitionen. Eine *Stelle* entspricht einer Zwischenablage für Daten, eine *Transition* beschreibt die Verarbeitung von Daten. Stellen werden durch Kreise ◯, Transitionen durch Balken ▬ dargestellt. Die Kanten dürfen jeweils nur von einer Sorte zur anderen führen. Die Stellen, von deren Kanten zu einer Transition t laufen, heißen *Eingabestellen* von t, andere Stellen, zu denen von t aus Kanten führen, heißen *Ausgabestellen* von t.

Der Graph spiegelt die feste Ablaufstruktur wider. Um dynamische Vorgänge zu beschreiben, werden die Stellen mit Objekten belegt, die durch die Transitionen weitergeleitet werden. Die Objekte sind Elemente eines Datentyps. Verwendet man boolsche Werte oder natürliche Zahlentypen als Datentypen, so werden diese als Marken bezeichnet. Der Bewegungsablauf der Marken im Petri- Netz wird durch folgende *Schaltregeln* festgelegt:

1. Eine Transition t kann schalten, wenn jede Eingabestelle von t mindestens eine Marke enthält.

2. Schaltet eine Transition, dann wird aus jeder Eingabestelle eine Marke entfernt und zu jeder Ausgabestelle eine Marke hinzugefügt.

Zu diesem Grundprinzip gibt es viele Modifizierungen. So kann z.B. jede Stelle S eine *Kapazität* $K(s)$ erhalten, die angibt, wie viele Marken höchstens in s liegen dürfen. Jede Kante kann aber auch eine Gewichtsfunktion erhalten, mit deren Hilfe man mehrere Marken gleichzeitig aus der Stelle abziehen und in Stellen fließen lassen kann.

Petri- Netze benutzt man zum Modellieren von Ablaufstrukturen, bei denen mehrere Prozesse und möglicherweise verschiedene Objekttypen auftreten, die nebeneinander

auftreten, die nebeneinander existieren oder in einer bestimmten Weise Daten miteinander austauschen oder miteinander um Hilfsmittel im Wettstreit stehen.

Typische Fragestellungen sind:*Terminiert* das Petri- Netz? Ist jede *Transition* lebendig? Das heißt, kann man die Transitionen stets so schalten, dass eine vorgegebene Transition t im weiteren Verlauf nochmal schalten kann? Treten vermeidbar Verklemmungen auf, Erreichbarkeitsproblem?

Formal definiert man ein Petri- Netz, welches mit natürlichen Zahlen arbeitet, als 5-Tupel $P = (S, T, A, E, M)$, mit:

- S ist eine nichtleere Menge von Stellen.
- T ist eine nichtleere Menge von Transitionen.
- $S \cap T = \emptyset$
- $A \subset S \times T$ ist eine Menge von Kanten, die von Stellen ausgehen und zu einer Transition gehen
- $E \subset T \times S$ ist eine Menge von Kanten, die von Transitionen ausgehen und bei Stellen enden.
- $M : S \to \mathbb{N}_0$ ist eine Markierungsfunktion, die angibt, wieviele Marken sich in jeder Stelle befinden.

Anhang E

Veröffentlichungen

Teile der in Kapitel 3 vorgestellten Ergebnisse sind in folgenden Arbeiten veröffentlicht worden:

- J. Zhang and Y. v. Collani and A. Knoll.
  Development of a Robot Agent for Interactive Assembly.
  In Proceedings of the 4th International Symposium on Distributed Robotic Systems, Karlsruhe, 1998.

- J. Zhang, Y. v. Collani and A. Knoll
  Interactive Assembly by a Two-Arm Robot Agent.
  Journal of Robotics and Autonomous Systems, Elsevier Science, 1999.

Teile der in Kapitel 4 vorgestellten Ergebnisse sind in folgenden Arbeiten veröffentlicht worden:

- Y. von Collani and J. Zhang and A. Knoll.
  A Neuro-Fuzzy Solution for Fine-Motion Control Based on Vision and Force Sensors.
  In Proceedings of the IEEE International Conference on Robotics and Automation, Leuven, Belgium, 1998.

- J. Zhang and Y. v. Collani and A. Knoll.
  Development of a Robot Agent for Interactive Assembly.
  In Proceedings of the 4th International Symposium on Distributed Robotic Systems, Karlsruhe, 1998.

- J. Zhang, Y. v. Collani and A. Knoll.
  Interactive Assembly by a Two-Arm Robot Agent.
  Journal of Robotics and Autonomous Systems, Elsevier Science, 1999.

- Y.v. Collani, C. Scheering, J. Zhang, A. Knoll.
  A neuro-fuzzy solution for integrated visual and force control.

In Proceedings of the International Conference on Multisensor Fusion and Integration of Intelligent Systems, Taipeh, Aug. 1999.

- Y. von Collani, M. Ferch, J. Zhang and A. Knoll.
  A General Learning Approach to Multisensor Based Control using Statistical Indices. Proc. 2000 IEEE Conf. on Robotics and Automation, San Francisco, California, April 2000

- J. Zhang, Y. v. Collani, and A. Knoll.
  A learning-based multisensor fusion approach for fine motion control of robot arms.
  In Proceedings of Robotik 2000, pages 167–172, Berlin, 2000.

- Y. v. Collani and J. Zhang and A. Knoll
  A Generic Learning Approach to Multisensor Based Control.
  Submitteld to International Conference on Multisensor Fusion and Integration of Intelligent Systems, Baden-Baden , 2001.

Abbildungsverzeichnis

1.1 *Verwendete Bauteile und angestrebtes Zielaggregat des SFB's 360.* . . . 10

2.1 *Hierarchiespektrum.* . . . 14
2.2 *Ein typischer Knoten in der* Real-time Control System *(RCS) Architektur (aus [Alb97a]).* . . . 16
2.3 *Hierarchische Architektur (aus [Alb97a]).* . . . 17
2.4 *Direkter Graph einer Montagesequenz mit den Bauteilen C,S,R,H (aus [RW91]).* . . . 23
2.5 *AND/OR Graph für die Produktion von ABCDE.* . . . 24
2.6 *Beispiel für einen Montagegraphen in Form eines Petri-Netzes.* . . . 26

3.1 *Schema des Gesamtsystems. Das Kernsystem besteht nur aus der Benutzerschnittstelle, einem Skriptinterpreter, dem* Blackboard *und dem Modulmanagement. Alle weiteren Funktionalitäten müssen über Module realisiert werden.* . . . 34
3.2 *Kommunikationsverbindungen eines Moduls.* . . . 35
3.3 *Interkommunikationsmöglichkeiten von Modulen.* . . . 36
3.4 *Schnittstelle zu einem realen Roboter. Ein Modul kommuniziert nicht direkt mit der Robotersteuerungssoftware, sondern bedient sich einer Zwischenschicht, die die Anweisungen an die Steuersoftware weiterreicht.* . . . 37
3.5 *Kommunikationsverbindungen eines Moduls zum Roboter. Ein Modul bekommt über das Robotermanagment die notwendigen Informationen zur Steuerung der Roboter.* . . . 38
3.6 *Schnittstelle zu einem virtuellen Roboter. Die Informationen über die Anzahl der Roboter muss nicht unbedingt mit der realen Anzahl übereinstimmen. Ein Anwendung kann z.B. nur Informationen über zwei Roboter besitzen, obwohl die Steueranweisungen auf vier reale Roboter umgesetzt werden.* . . . 39
3.7 *Steuerung von Manipulatoren deren Steuersoftware auf mehrere Steuerrechner verteilt ist. Die Steueranweisungen an den Roboter werden von der Hardwareabstraktionsschicht nicht direkt an die Robotersteuerungssoftware übergeben. Da die Roboter von unterschiedlichen Rechnern kontrolliert werden, werden die Anweisungen erst an diese Rechner gesendet und dort von der Steuersoftware umgesetzt.* . . . 40

3.8 *Beispielskript als Flussdiagramm.* 44
3.9 *Beispielsequenz mit* Recover- *und* Catch-*Umgebung als Fußdiagramm. Die gestrichelten Kanten geben den möglichen Kontrollfluss beim Auftreten eines Ereignisses an.* 46
3.10 *Informationsfluss bei direkter Anforderungen von Sensordaten. Ein Modul fordert über einen direkten Funktionsaufruf von einem Sensormodul dessen Daten. Das Sensormodul ermittelt die Sensordaten und liefert sie dem aufrufenden Modul in einer Datenstruktur zurück.* 48
3.11 *Informationsfluss bei Anforderung von Sensordaten über das Blackboard. Statt die Daten dem aufrufenden Modul zu übermitteln werden die Sensordaten im Blackboard abgelegt. Dort kann ein Sensormodul die Daten auch periodisch aktualisieren, ohne explizite Aufforderung.* 49
3.12 *Montagezelle mit zwei Manipulatoren.* 50
3.13 *Aufbau der Montagezelle.* 51

4.1 *B-Spline Basisfunktionen mit unterschiedlicher Ordnung.* 54
4.2 *B-Splines der Ordnung zwei und drei.* 55
4.3 *B-Spline Basisfunktionen zum Knotenvektor ξ und ihre Summe $\sum_{i=0}^{k-n} N_i^n(x)$.* 56
4.4 *Mögliche Ausgabekurve eines Fuzzy-Reglers mit B-Funktionen als ZF'n zur Regelung der Andruckkraft während eines Schraubvorganges.* 57
4.5 *Exemplarische Sensordatenfusion.* 59
4.6 *Die sortierten Eigenwerte einer durch Bilder gebildeten Kovarianzmatrix.* 60
4.7 Sichten einer Handkamera während der Steuerungsoperation (Bildgröße: 111×103). Die Bilder zeigen das Loch in der Leiste. Durch das Loch ist die Schraubenspitze zu sehen, die über dem Loch zentriert werden soll. Teilweise sind ebenfalls Teile des Manipulators zu sehen, der die Schraube hält (z.B. die Handkamera in Bild (a)). 61
4.8 *Die Struktur eines Fuzzy Reglers mit vorgeschalteter Reduktion über PCA.* 62
4.9 *Fusion mehrerer unterschiedlicher Sensordaten.* 67
4.10 *Schraubvorgang mit einer langen Schraube.* 68
4.11 *Verschiedene Stellungen einer Schraube in einem Gewindeloch.* 69
4.12 *Fusion zweier Ansichten ein und derselben Szene.* 69
4.13 *Rücktransformation von Eigenvektoren in Graustufenbilder.* 70
4.14 Rücktransformation des über ORF gewonnenen Projektionsvektors (a) für die Auslenkung um den Normalenvektor, (b) für die um den Schließvektor). 71
4.15 *Sollwinkel gegen Projektion der Bilder von der Draufsicht.* 72
4.16 *Sollwinkel gegen Projektion der Bilder von der Seitansicht.* 74

5.1 *Flussdiagramm einer Beispielsequenz.* 78
5.2 *Beispielsequenz aus Abb. 5.1 als direkter Montagegraph.* 80
5.3 *Darstellung der modellierten Bauteile.* 81

5.4 *Beispielvisualisierung eines internen Zustandes. Die Abbildung zeigt mehrere auf dem Montagetisch liegende Bauteile und eine Leiste und eine Schraube, die von den (nicht dargestellten) Robotern gehalten werden.* . 82
5.5 *Sensordatenaquirierung.* 83
5.6 *Flussdiagramm einer Beispielsequenz mit* Recover*-Abschnitt.* 85
5.7 *Flussdiagramm einer Beispielsequenz zum Speichern der Position und des Greiferzustandes.* 86
5.8 *Flussdiagramm einer Beispielsequenz zum Speichern der Position und des Greiferzustandes inkl.* Recover*-Abschnitt. Die Instruktionen zur Aktualisierung der internen Repräsentation sind zur besseren Übersicht in den Flussdiagrammen nicht dargestellt.* 88

6.1 *Flussdiagramm zweier Sequenzen zum Bau eines Leitwerks.* 89
6.2 *Beispiel für zusätzliche Instruktionen.* Sz_i *und* Sq_i *repräsentieren die aufgezeichneten Sensorzustände der* i*-ten Instruktion.* 92
6.3 *Beispiel zweier unterschiedlicher Teilsequenzen (fett).* Sz_i *und* Sq_i *repräsentieren die aufgezeichneten Sensorzustände der* i*-ten Instruktion.* . 93
6.4 *Erfüllung der Bedingung (6.3).* Sz_i *und* Sq_i *repräsentieren die aufgezeichneten Sensorzustände der* i*-ten Instruktion.* 94
6.5 *Erfüllung der Bedingung (6.5).* Sz_i *und* Sq_i *repräsentieren die aufgezeichneten Sensorzustände der* i*-ten Instruktion.* 95
6.6 Teilabschnitt der neuen Zielsequenz η als Flussdiagramm (Fall 1). 97
6.7 *Flussdiagramm der resultierende Sequenz für Fall 1.* Sz_i *und* Sq_i *repräsentieren die abgelegten Sensorzustände der* i*-ten Instruktion. Es ist zu erkennen, wie die aufgezeichneten Sensordaten nach der Zusammenlegung von zwei Sequenzen zugeordnet werden. Der* IF*-Instruktion werden die Sensordaten der möglichen Nachfolgeinstruktion zugeordnet, da beide Zustände vor dieser Verzweigung aufgetreten sind.* 98
6.8 Teilabschnitt der neue Zielsequenz η als Flussdiagramm (Fall 3). 99
6.9 *Flussdiagramm der resultierende Sequenz für Fall 3.* Sz_i *und* Sq_i *repräsentieren die abgelegten Sensorzustände der* i*-ten Instruktion. Es ist zu erkennen, wie die aufgezeichneten Sensordaten nach der Zusammenlegung von zwei Sequenzen zugeordnet werden. Der* IF*-Instruktion werden die Sensordaten der möglichen Nachfolgeinstruktion zugeordnet, da beide Zustände vor dieser Verzweigung aufgetreten sind.* 100
6.10 *Beispiel für Differenzen aufgrund von Verzweigungen.* 101
6.11 *Für die Berechnung der boolschen Funktion betrachtete Instruktionen (fett).* Sz_i *und* Sq_i *repräsentieren die abgelegten Sensorzustände der* i*-ten Instruktion. Dir Sensordaten der Zustände* Sz_2 *und* Sq_2 *werden für die Berechnung der boolschen Funktion herangezogen.* 102
6.12 Beispielsituation vor dem Ausrichten einer Leiste. 104
6.13 Reduktion der Sensordaten von der Handkamera. 106
6.14 *Aufbau des Entscheidungcontrollers.* 106

6.15 *Flussdiagramm der Beispielsequenz zum Bau eines Propellers.* 108
6.16 *Flussdiagramm der Beispielsequenz zum Bau eines Propellers.* 109
6.17 *Vereinfachter Ausschnitt einer Rumpfbausequenz.* 110
6.18 *Vereinfachter Ausschnitt einer Rumpfbausequenz mit Verzweigung der Handlung.* 111

7.1 *Aggregate aus dem Spielzeug* Baufix*.* 115
7.2 *Rumpf des Baufix-Flugzeuges.* 116
7.3 *Verschiedene Zustände während der Montage.* 117
7.4 *Flussdiagramm zweier Sequenzen zum Bau des Leitwerks. Die Instruktionen zur Aktualisierung der internen Repräsentation sind zur besseren Übersicht in den Flussdiagrammen nicht dargestellt.* 118
7.5 *Reihenfolge der Bearbeitung einer Instrukteursanweisung. (1) Entgegennahme der Anweisung über die Benutzerschnittstelle und Umsetzen in einen Skriptaufruf. (2) Speicherung der aktuellen Sensordaten. (3) Übergabe des Skriptes an den Skriptinterpreter zur Ausführung. (4) Rückmeldung des Skriptinterpreter über den Erfolg der Ausführung und (5) Speicherung der Skriptaufrufen mit Sensordaten im Erfolgsfall.* 119
7.6 *Flussdiagramm der 1. Teilsequenz (Leitwerk) mit Sensordaten. Die Instruktionen zur Aktualisierung der internen Repräsentation sind zur besseren Übersicht in den Flussdiagrammen nicht dargestellt.* 120
7.7 *Flussdiagramm 2. Teilsequenz (Leitwerk) mit Sensordaten. Die Instruktionen zur Aktualisierung der internen Repräsentation sind zur besseren Übersicht in den Flussdiagrammen nicht dargestellt.* 121
7.8 *Ausgabefunktion des trainierten B-Spline Fuzzy Reglers.* 123
7.9 *Visualisierung des Projektionsvektors.* 124
7.10 *Zusammengefügte Sequenz zum Bau eines Leitwerks. Die Instruktionen zur Aktualisierung der internen Repräsentation sind zur besseren Übersicht in den Flussdiagrammen nicht dargestellt.* 128
7.11 *Vollständiges Flussdiagramm der zusammengefügten Sequenz zum Bau eines Rumpfes.* 129

8.1 *Architektur des Situierten Künstlichen Kommunikators aus Sicht der Aktorik.* 133
8.2 *OPERA als verteiltes System.* 134

A.1 *Hauptfenster von* OPERA *(Hauptmenü (1), Erweitertes Menü (2), Roboterstatus (3), aktuelle Sequenz (4), Meldungsfenster (5), aktuelle Bildanzeige (6)).* 140
A.2 *Sensorfenster von* OPERA*. Die Abbildung zeigt die visualisierten Daten von zwei Kraftmomentensensoren, deren Werte über die Zeit aufgetragen sind.* 141
A.3 *Dialog für die Sequenzparameter.* 142

A.4 *Dialog zur Dateiauswahl.* 142
A.5 *Dialog zur Roboterauswahl.* 143
A.6 *Liste aller geladenen Module.* 144
A.7 *Parameterangabe über einen Textstring.* 144
A.8 *Sequenzfenster mit Beispielsequenz.* 146
A.9 *Dialog zum Erstellen einer* IF-THEN *Instruktion.* 147

Tabellenverzeichnis

4.1 Größter, ungünstigster und mittlerer quadratischer Fehler. Ist kein ungünstigster Fehler aufgeführt, so findet in dieser Sensorkonfiguration keine fehlervergrößernde Korrekturbewegung statt. 71
4.2 Größter, ungünstigster und mittlerer quadratischer Fehler bei der Dimensionsreduzierung durch ORF. Ist kein ungünstigster Fehler aufgeführt, so findet in dieser Sensorkonfiguration keine fehlervergrößernde Korrekturbewegung statt. 73
4.3 Größter, ungünstigster und mittlerer quadratischer Fehler bei der Dimensionsreduzierung durch PCA mit Hinzunahme von Kraftmomentendaten. Ist kein ungünstigster Fehler aufgeführt, so findet in dieser Sensorkonfiguration keine fehlervergrößernde Korrekturbewegung statt. 73
4.4 Größter, ungünstigster und mittlerer quadratischer Fehler bei der Dimensionsreduzierung durch ORF und Hinzunahme von Kraftmomentendaten. Ist kein ungünstigster Fehler aufgeführt, so findet in dieser Sensorkonfiguration keine fehlervergrößernde Korrekturbewegung statt. 75

6.1 Beispiel zur Auswertung von Sensoren. Nur diejenigen Sensoren werden für die Auswertung betrachtet, die auch in allen Beispielzuständen vorkommen. Bei *Kraft in Z* ist dies nicht der Fall. 103
6.2 Beispiel zur Auswertung von Sensoren. Nur diejenigen Sensoren werden für die Auswertung betrachtet, die eine Varianz aufweisen. Bei *Position Z* ist dies nicht der Fall. 103
6.3 Sensorzustände der einzelnen Situationen bzw. Beispiele. 105
6.4 Korrelation der einzelnen Sensordaten mit den Solldaten. 105

7.1 Sensordaten an der Verzweigung. 122
7.2 Korrelation der relevanten Sensordaten zur jeweiligen Verzweigung (1. Verzweigung). 122
7.3 Korrelation der relevanten Sensordaten zur jeweiligen Verzweigung (2. Verzweigung). 123
7.4 Korrelation der relevanten Sensordaten der Verzweigung. 127

Index

R, 32
T, 32
Σ, 32
r, 32
.operarc, 147

Anweisungen, 77
Anweisungen
  direkte, 77, 84
  komplex, 77
  komplexen, 84
Architekturen, 13
Architekturen
  deliberativ, 13
  Komplexitätsreduzierung, 15
  RCS, 15
  reaktiv, 14
  Sprachen, 17

B-Spline
  Basisfunktionen, 53
B-Splines, 53
B-Splines
  Differenzierbarkeit, 54
  Eigenschaften, 54
  Knotenvektor, 54
  MISO, 58
  Ordnung, 54
  Partition of Unity, 54, 55
  Sensordatenfusion, 59
  Singletons, 57, 58
  SISO, 56
  Zugehorigkeitsfunktionen, 55
Benutzeroberfläche, 139
Benutzeroberfläche
  Komponenten, 139
  Komponenten
    Hauptfenster, 139
    Sensorfenster, 140
  Menüelemente, 141
  Menüelemente
    File, 141
    Module, 144
    Run, 143
    Status, 143
  Sequenzfenster, 145
Blackboard, 32, 33

Datenfusion, 59, 67

Ereignis
  Instruktionssemantik, 45
Ereignisse, 45, 77
Events, 83

Ganze Zahlen, 32
Gedächtnis, 110
Gedächtnis
  episodisch, 111, 112
  prozedural, 112
  semantisch, 112, 113
Generalisierung, 89
Generalisierung
  Modifikation, 108
  Retrieval, 107
  Sensormusterauswertung, 100
  Vergleichsmethode, 90
  Verzweigungen, 92
  Verzweigungen
    Fall 1, 96
    Fall 2, 96
    Fall 3, 97

Zusammenfassen von Sequenzen, 90
Gesamtsystemzustand, 33

Handkamera, 48

Instrukteur, 10

Konfigurationsdatei, 147

Module, 34
Module
Kategorien, 35
Montagezelle, 48

Natürliche Zahlen, 32
Nullterminierte Zeichenketten, 32

OCCL, 167
OCCL
Kanäle, 168
TOCCLObject, 168, 169
OPERA, 31, 139, 167
OPERA
Basissystem, 33
Benutzerschnittstelle, 33
Blackboard, 33
Merkmale, 31
Modulemanagement, 36
Modulmanagement, 33
Sensorzugang, 47
Skriptinterpreter, 31, 33
ORF, 66

PCA, 60
PCA
Berechnung, 61
Petri-Netz, 177
Programmierung, 147
Programmierung
Blackboard, 157
Compiler, 165
Methoden, 148
Methoden
AllocX, 152
CreateModuleInstance, 148
DeallocX, 152
Edit, 151
GetCommandName, 148
GetParameterDescription, 152
GetTextFromParamString, 152
GetType, 149
operator(), 149
Roboter Modul, 156
TModuleDatabase, 159
TParameterList, 159
TRemoteRobot, 160
TVariableDatabase, 157
Programming by Demonstration, 28

Reelle Zahlen, 32
Repräsentation, 18, 79
Repräsentation
AND/OR, 22
direkt, 22
intern, 80
Montageoperationen, 20
Montageprozess, 19
Montagezustände, 20
Petri-Netz, 25
Sequenzen, 21
Verbindungen, 19
Verbindungsbedingungen, 24
Roboterzustand, 32

Sensordatenfusion, 59
Sensorfenster, 140
Sensormodul, 47
Sensorrepräsentation, 53
Sensortrajektorie, 82
Sensorzustand, 33
Sequenzen, 41
Skriptinterpreter, 41
Skriptinterpreter
Instruktionssemantik, 42
Programmsemantik, 43
Sonderforschungsbereich 360, 133
Sonderforschungsbereich 360
Architektur, 134

TOCCLObject, 32, 168, 169

Virtueller Roboter, 38

Literaturverzeichnis

[AAB+91] T.E. Abell, G.P. Amblard, D.F. Baldwin, T.L. De Fazio, Man-Cheung M. Lui, and D.E. Whitney. *Computer-Aided Mechanical Assembly Planing*, chapter Computer aids for finding, representing, choosing amongst, and evaluating the assembly sequences of mechanical products, pages 383–435. Kluwer Academic Press Boston, 1991.

[AdA98] R. Araujo and Anibal T. de Almeida. Map building using fuzzy art and learning to navigate a mobile robot on an unkown world. In *In Proceedings of the IEEE Int. Conf. on Robotics and Automation*, 1998.

[AI89] H. Asada and H. Izumi. Automatic program generation from teaching data for the hybrid control of robots. *IEEE Trans. on Robotcs and Automation*, 5(2):163–173, April 1989.

[Alb95] J. S. Albus. The nist real- time control system (rcs): An application survey. In *Proceedings of the AAAI Spring Symposium Series*, 1995.

[Alb97a] J.S. Albus. 4-d/rcs: A reference model architecture for demo iii, version 1.0. Technical report, National Institute of Standards abd Technology, Gaithersburg,MD, March 1997.

[Alb97b] J.S. Albus. The nist real-time control system (rcs): an approach to intelligent systems design. *Journal of Experimental and Theoretical Artifical Intelligence*, 9(2-3):157–174, 1997.

[Alb00] James S. Albus. 4-d/rcs reference model architecture for unmanned ground vehicles. In *Proceedings of the 2000 IEEE Intrl. Conf. on Robotics and Automation*, pages 3260–3265, April 2000.

[Ark87] R.C. Arkin. Motor schema based navigation for a mobil robot: An approach to programming behavior. In *Proc. of the Int. Conf. on Robotics and Automation*, pages 264–71, 1987.

[Ark98] R.C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

[BA01] D. Bentivegna and C. Atkeson. Learning from observation using primitives. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, pages 1988–1993, 2001.

[BKS99] Christian Baukhage, Franz Kummert, and Gerd Sagerer. Learning assembly sequence plans using functional models. In *International Symposium on Assembly and Task Planning*, ISART, pages 1–7. IEEE, 1999.

[BKW97] R. Peter Bonasso, David Kortenkamp, and Troy Whitney. Using a robot control architecture to automate space shuttle operations. *AAAI*, 1997.

[Bro86] R.A. Brooks. A robust layered control system for mobile robot. *IEEE Journal of Robotiks and Automation*, 2(1):14–23, March 1986.

[CG87] G.A. Carpenter and S. Grossberg. A massively parallel architecture for a selforganizing neuronal pattern regonition machine. *Computer Vision, Graphics and Image Processing*, 37:54–115, 1987.

[Chi96] S. L. Chiu. Selecting input variables for fuzzy models. *Journal of Intelligent and Fuzzy Systems*, 4:243–256, 1996.

[CM98] J. Chen and B. McCarragher. Robot programming by demonstration - selecting optimal event path. In *Proceedings of the IEEE Int. Conf. Robotics and Automation*, 1998.

[CM00] J.R. Chen and B.J. McCarragher. Programming by demonstration - constructing task level plans in a hybrid dynamic framework. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, ICRA, pages 1402–1407, April 2000.

[CMS00] Eve Coste-Maniere and Reid Simmons. Architecure, the backbone of robotics systems. In *In Proceedings of the IEEE Intl. Conf. on Robotics and Automation*, April 2000.

[Con89] Connell. A behaviour-based arm controller. *IEEE Transaction on Robotics and Automation*, 5(6):784–791, 1989.

[CS91] T. Cao and A.C. Sanderson. Task sequence planning in a robot work cell using AND/OR nets. In *Proceedings of the 1991 IEEE International Symposium on Intelligent Control*, pages 239–244, Aug. 1991.

[CS98] Tiehua Cao and Artur C. Sanderson. AND/OR net representation for robotic task sequence planning. *IEEE Transactions on Systems, Man, and Cybernetics-Part C:*, 28(2):204–217, May 1998.

[dM89] L. Homem de Mello. *Task Sequence Planning for Robotic Assembly*. PhD thesis, Carnegie Mellon, 1989.

[DOC] *FLTK 1.0.10 Programming Manual*. http://www.fltk.org/doc/toc.html.

[ea93] T. Suzuki et al. On algebraic and graph structural properties of assembly petri net. In *Proceedings of the 1993 IEEE International Conference on Robotics & Automation*, pages 794–800, 1993.

[ea99] J.M. Roberts et al. A real-time software architekture for robotics and automation. In *In Proceedings of the IEEE Int. Conf. on Robotics and Automation*, 1999.

[EMR96] Engeln-Muellges and Reutter. *Numerik- Algorithmen*. VDI Verlag, 1996.

[Eng88] Hermann Engesser. *Duden Informatik*. Dudenverlag, 1988.

[Eys94] M. W. Eysenck. *Dictionray of Cognitive Psychology*. 1994.

[FBW01] B. Finkemeyer, M. Borchard, and F. Wahl. A robot control architecture based on an object server. In *International Conference Robotics and Manufacturing*, 2001.

[Fer01] Markus Ferch. *Lernen von Montagestrategien in einer Multiroboterumgebung*. PhD thesis, University of Bielefeld, 2001.

[FG98] J.A. Fernandez and J. Gonzalez. Nexus: A flexible, efficient and robust framework for integrating software components of a robotic system. In *Proceedings of the IEEE International Conf. on Robotics and Automation, Leuven, Belgium*, ICRA, May 1998.

[FHD98] H. Friedrich, J. Holle, and R. Dillmann. Interactive generation of flexible robot programs. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, ICRA, pages 538–543, May 1998.

[Fir89] Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, New Haven, C.T., 1989.

[FNUH01] T. Fukuda, M. Nakaoka, T. Ueyama, and Y. Hasegawa. Direct teaching and error recovery method for assembly task based on a transition process of a constraint condition. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, pages 1518–1523, 2001.

[FZK99] M. Ferch, J. Zhang, and A. Knoll. Robot skill transfer based on b-spline fuzzy controller for force-control tasks. In *In Proceedings of the IEEE International Conference on Robotics and Automation, Detroit*, ICRA, 1999.

[Gat92] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *In Tenth National Conf. on Artificial Intelligence*, July 1992.

[Gat96] E. Gat. Esl: A language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the AAAI Fall Symposium on Plan Execution*, 1996.

[GR74] W.J. Gordon and R.F. Riesenfeld. *B-spline curves and surface.* Computer Aided Geometric Design, Academic Press, 1974.

[HM99] C. Hiransoog and C.A. Malcolm. Multi-sensor / knowledge fusion. In *Proceedings of the Intl. Conf. on Multi sensor Fusion and Integration for Intelligent Systems*, pages 117–122, Aug. 1999.

[HP86] V. Hayward and R. Paul. Robot manipulator control under Unix RCCL: A robot control "C" library. *Int. Journal of Robotics Research*, 1986.

[JM98] C.G. Johnson and D. Marsh. A robot programming environment based on free-form CAD modelling. In *Proceedings of the IEEE International Conf. on Robotics and Automation, Leuven, Belgium*, ICRA, May 1998.

[JSM97] J.S.R. Jang, C.T. Sun, and E. Mizutani. *Neuro- Fuzzy and Soft Computing*. Prentice Hall, 1997.

[JW96] R. E. Jones and R.H. Wilson. A survey of constraints in automated assembly planning. In *Proceedings of the Int. Conf. on Robotics and Automation*, pages 1525–1532, 1996.

[Kae86] L. Kaelbing. An architecture for intelligent reaktive systems. Technical Report 400, SRI, October 1986.

[Kai96] Michael Kaiser. *Interaktive Akquisition elementarer Roboterfhigkeiten*. PhD thesis, Univ. Karlsruhe, 1996.

[Kla90] Herbert Klaeren. *Vom Problem zum Programm.* B.G. Teubner Stuttgart, 1990.

[LT97] V. Lacrose and A. Tilti. Fusion and hierarchy can help fuzzy logic controller designers. In *Proceedings of the IEEE Int. Conf. on Fuzzy Systems*, 1997.

[Mae90] Maes. Situated agents can have goals. *Robotics and Autonomous Systems*, 6:44–70, 1990.

[Mos00] Heiko Mosemann. *Beiträge zur Planung, Dekomposition und Ausführung von automatischen generierten Roboteraufgaben.* PhD thesis, Tech. Univ. Braunschweig, 2000.

[MPB01] D.R. Myers, M.J. Pritchard, and M.D.J. Brown. Automated programming of an industrial robot through teach-by showing. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, pages 4078–4083, 2001.

[MRW97] H. Mosemann, F. Röhrdanz, and F. Wahl. *Robotics Research: The Eighth International Symposium*. Springer, 1997.

[MRW98] H. Mosemann, F. Röhrdanz, and F. Wahl. Assembly stability as a constraint for assembly sequence planning. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, ICRA, 1998.

[Mye99] Donald R. Myers. An approach to automated programming of industrial robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, ICRA, pages 3109–3114, May 1999.

[Nil80] Nils J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag Berlin, Heidelberg, New York, 1980.

[NMN94] S. K. Nayar, H. Murase, and S.A. Nene. Learning, positioning and tracking visual appearance. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, ICRA, pages 3237–3244, 1994.

[PHH99] J. Peterson, G.D. Hager, and P. Hudak. A language for declarative robotic programming. In *In Proceedings of the IEEE Int. Conf. on Robotics and Automation*, 1999.

[Pro93] Adept Technology. *V^+ User's Guide v.11*, 1993.

[Qui66] M. R. Quillian. *Semantic memory*. PhD thesis, Carnegie Insitute of Technologie, 1966.

[RFG+97] R.P.Bonasso, R.J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. A proven three-tiered architectur for programming autonomous robots. *Journal of Experimental and Theoretical Artifical Intelligence*, 1997.

[Rie73] R.F. Riesenfeld. *Applications of B-Spline approximation to geometric problems of computer-aided design*. PhD thesis, Syracuse University, 1973.

[RMW97] F. Röhrdanz, H. Mosemann, and F. Wahl. Generating and evaluating stable assembly sequences. *Advanced Robotics*, 11(2):97–126, 1997.

[RW91] A.A.G. Requicha and T.W. Whalen. *Computer-aided mechanical assembly planning*, chapter Representationsfor Assemblies, pages 15–39. Kluwer Academic Press Boston, 1991.

[SA98] R. Simmons and D. Apfelbaum. A task description language for robot control. In *In Proceedings of the Int. Conf. on Intelligent Robots and Systems*, 1998.

[Sch46] I.J. Schoenberg. *Contributions to the problem of approximation of equidistant data by analytic functions*. Quarterly of Applied Mathematics, 1946.

[Sch98a] Ralf Schmidt. Ein robustes Verfahren zur Roboter- Feinpositionierung durch visuelles Lernen. Master's thesis, University of Bielefeld, Faculty of Technology, December 1998.

[Sch98b] Volkmar Schwert. Entwicklung eines selbstlernenden Fuzzy-Systems zu autonomen Navigation und Lokalisierung eines mobilen Roboters. Master's thesis, University of Bielefeld, Faculty of Technology, May 1998.

[Sch00a] Christian Scheering. *Multi-Agenten in einem situierten künstlichen Kommunikator*. PhD thesis, University of Bielefeld, 2000.

[Sch00b] Stefan Schlosske. Development of a modular robot controll. Master's thesis, University of Bielefeld, April 2000.

[SCPCW98] S. Schneider, V. Chen, G. Pardo-Castellote, and H. Wang. Controlshell: A software architecture for complex electro-mechanical systems. *International Journal of Robotics Research*, 1998.

[SD94] S.U. Seow and R. Devanathan. A temporal framework for assembly sequences representation and analysis. *IEEE Trans. Robot. Automation*, 10(2):220–229, 1994.

[Smi92] Michael G. Smith. An environment for more easily programming a robot. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, pages 10–16, 1992.

[SRC01] P. Sebastiani, M. Ramoni, and P. Chohen. *Sequenz Learning*, chapter Sequenz learning via Bayesian Clustering by Dynamics. Springer, 2001.

[TAG01] A. Traub, A. Ayat, and B. Graf. Ein komponentenbasierte Softwarearchitektur zur Steuerungsentwicklung. In *A+D Compendium*. 2001.

[TKC+94] E. Tulving, S. Kapur, F.I.M. Craik, M. Moscovitch, and S. Houle. Hemispheric encoding/retrieval asymmetry in episodic memory: Positron emission tomography findings. In *Proceedings Nat. Acad. Sci.*, number 91, pages 2016–2020, 1994.

[TNB96] Johnson P. Thomas, Nimal Nissanke, and Keith D. Baker. A hierarchical petri net framework for the representation and analysis of assembly. In *IEEE Transactions on Robotics and Automation*, volume 12, pages 268–279, April 1996.

[TS99] A. Traub and R.D. Schraft. An object-oriented realtime framework for distributed control systems. In *In Proceedings of the IEEE Int. Conf. on Robotics and Automation*, 1999.

[TTO+99] H. Tominaga, J. Takamatsu, K. Ogawara, H. Kimura, and K. Ikeuchi. Symbolic representation of trajectories for skill generation. In *In Proceedings of the IEEE Int. Conf. on Robotics and Automation*, 1999.

[Tul83] E. Tulving. *Elements of Episodic Memory*. Clarendon Press, Oxford, 1983.

[Tul85] E. Tulving. How many memory systems are there ? *American Psychologist*, 40:385–398, 1985.

[vCFZK00] Y. von Collani, M. Ferch, J. Zhang, and A. Knoll. A general learning approach to multi sensor based control using statistical indices. In *Proc. of the IEEE Int. Conf. on Robotics and Automation, San Francisco, California*, April 2000.

[vCSZK99] Y. von Collani, C. Scheering, J. Zhang, and A. Knoll. A neuro-fuzzy solution for integrated visual and force control. In *Proceedings of the International Conference on Multi sensor Fusion and Integration of Intelligent Systems, Taipeh*, Aug. 1999.

[vCZK98] Y. von Collani, J. Zhang, and A. Knoll. A neuro-fuzzy solution for fine-motion control based on vision and force sensors. In *Proceedings of the IEEE International Conference on Robotics and Automation, Leuven, Belgium*, ICRA, 1998.

[Wal97] J. Walter. Sorma: Inter operating distributed robotics hardware. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, ICRA, 1997.

[WDMA96] J.R. Whiteley, J.F. Davis, A. Mehrota, and S.C. Ahalt. Observations and problems applying art2 for dynamic sensor pattern interpretation. *IEEE Transactions on Systems, man and cybernetics*, 26(4), July 1996.

[Wei87] L. L. Weiskrantz. Neuroanatomy of memory and amnesia. *Neurobiology*, 6:93–105, 1987.

[WRS99] Shige Wang, C.V. Ravishankar, and Kang G. Shin. Open architecture controller software for integration of machine tool monitoring. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, pages 1152–1157, 1999.

[WS98] Q. Wang and J. De Schutter. Towards real-time robot programming by human demonstration for 6d force controlled actions. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, pages 2256–2261, 1998.

[WS01] S. Wang and K.S. Shin. Reconfigurable software for open architecure controllers. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, pages 4090–4095, 2001.

[YKC89] A.L. Yuille, D.M. Kammen, and D.S. Chohen. Quadrature and the development of orientation selective cortical cells by hebb rules. *Biological Cybernetics*, 61, 1989.

[ZF98] J. Zhang and M. Ferch. Rapid on-line learning of compliant motion for two-arm coordination. In *Proceedings of the IEEE International Conference on Robotics and Automation, Leuven, Belgium*, ICRA, 1998.

[ZFK00] J. Zhang, M. Ferch, and Alois Knoll. Carrying heavy objects by multiple manipulators with self-adapting force control. In *Proc. of Intelligent Autonomous Systems*, pages 204–211, 2000.

[ZK96] J. Zhang and A. Knoll. Constructing fuzzy controllers with b-spline models. In *Proceedings of IEEE International Conference on Fuzzy Systems*, 1996.

[ZKS99] J. Zhang, A. Knoll, and R. Schmidt. A neuro-fuzzy control model for fine-positioning of manipulators. *Journal of Robotics and Autonomous Systems, Elsevier Science*, 1999.

[ZKS00] J. Zhang, A. Knoll, and R. Schmidt. A neuro fuzzy control model for fine-positioning of manipulators. *Journal of Robotics and Autonomous System*, 32:101–113, 2000.

[ZL96] J. Zhang and K.V. Le. Self-optimization of a fuzzy controller with b-spline models. In *Proceedings of Fourth European Congress on Intelligent Techniques and Soft Computing*, 1996.

[ZMS97] D. Zühlke, F. Möbius, and C. Schröder. Symbols facilitate programming of industrial robots. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, ICRA, 1997.

[ZRH94] J. Zhang, J. Raczkowsky, and H. Herp. Emulation of spline curves and its application in robot motion control. In *Proceedings of IEEE International Conference on Fuzzy Systems*, pages 831–836, 1994.

[ZSK99] J. Zhang, R. Schmidt, and A. Knoll. Appearance-based visual learning in a neuro-fuzzy model for fine-positioning of manipulators. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1999.

[ZvCK97] J. Zhang, Y. von Collani, and A. Knoll. On-line learning of b-spline fuzzy controller to acquire sensor-based assembly skills. In *Proc. of the IEEE Conf. on Robotics and Automation, Albuquerque, New Mexico*, ICRA, April 1997.